1-1-2002

Subsynth: A generic audio synthesis framework for real-time applications

Kevin August Meinert
Iowa State University

Subsynth: A generic audio synthesis framework for real-time applications

by

Kevin August Meinert

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Carolina Cruz-Neira, Major Professor
Daniel Ashlock
Anne Deane
Julie Dickerson
Govindarasu Manimaran

Iowa State University

Ames, Iowa

2002

Graduate College
Iowa State University

This is to certify that the master's thesis of

Kevin August Meinert

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF LISTINGS vii

ACKNOWLEDGEMENTS viii

ABSTRACT ix

CHAPTER 1 INTRODUCTION 1

RESEARCH PROBLEM 1
STATEMENT OF PURPOSE 2
SCOPE OF RESEARCH 3

CHAPTER 2 BACKGROUND 6

DIGITAL AUDIO 6
Sampling 7
Quantizing 8
Digital/Analog Conversion 9
Dual Representation 10
Distortion 11
COMMON AUDIO FORMATS USED IN DIGITAL COMPUTERS TODAY 12
DIGITAL AUDIO SYNTHESIS 18
Unit Generator Language 19
Unit Generator Building Blocks 20
Fixed-Waveform Table-lookup Synthesis 23
Additive Synthesis and Wavestacking 26
Subtractive Synthesis 26
Non-linear or Modulation Synthesis 27
SONIFICATION 28
Motivation for Sonification 29
Examples of Sonification 30
When to Use Sonification Instead of Visualization 31
Requirements for Sonification Tools 31
Methods of Sonification 32

CHAPTER 3 REQUIREMENTS OF AN AUDIO SUBSYSTEM FOR INTERACTIVE APPLICATIONS 34

SCIENTIFIC SONIFICATION 34
INTERACTIVE COMPOSITION AND PERFORMANCE 34
VIRTUAL ENVIRONMENT SONIFICATION 35
REQUIREMENTS 36

CHAPTER 4 CURRENT TOOLS FOR INTERACTIVE AUDIO 40

MUSICAL INSTRUMENT DIGITAL INTERFACE (MIDI) 40

CHAPTER 4 CURRENT TOOLS FOR INTERACTIVE AUDIO 40
MUSICAL INSTRUMENT DIGITAL INTERFACE (MIDI) 40
CSOUND 42
SUPERCOLLIDER 43
VIRTUAL AUDIO SERVER (VAS) 45
VIRTUAL SOUND SERVER (VSS) 46
DIGITAL INSTRUMENT FOR ADDITIVE SOUND SYNTHESIS (DIASS) 48
OPENAL 50
PORTAUDIO/PABLIO 52
CHAPTER 5 SUBSYNTH 56
CONCEPTUAL DESIGN APPROACH 57
Core Audio Framework 59
Configuration 61
Task Management System 62
IMPLEMENTATION 63
Technology Choices 63
Core Audio Framework 64
Included Modules 71
Included Builders 74
Included Runners 76
CHAPTER 6 CASE STUDIES 80
SUBSYNTHMIDI 80
G.A.M.E. TOOLKIT FOR SCIENTIFIC SONIFICATION 82
L-System Background 83
G.A.M.E. Music Engine Implementation 84
SONIX 85
Reconfiguration 86
Using Sonix 86
Sonix Architecture and Design 88
CHAPTER 7 DISCUSSION OF RESULTS 93
BENEFITS 93
LIMITATIONS 94
CHAPTER 8 CONCLUSION 95
CHAPTER 9 FUTURE WORK 97
Appendix A 100
Appendix B 103
Appendix C 105
BIBLIOGRAPHY 107

LIST OF FIGURES

Number

FIGURE 1. SAMPLING A 770HZ SINUSOID AT 700HZ.....7

FIGURE 2. ERRORS DURING QUANTIZATION OF A SMOOTH CURVE.....9

FIGURE 3 TIME DOMAIN REPRESENTATION OF A SIGNAL.....10

FIGURE 4. FREQUENCY DOMAIN REPRESENTATION OF A SIGNAL.....10

FIGURE 5. SINGLE AUDIO CHANNEL OF PCM DATA IN MEMORY.....13

FIGURE 6. NONINTERLACED PCM AUDIO DATA WITH N NUMBER OF CHANNELS.....14

FIGURE 7. INTERLACED PCM AUDIO DATA.....14

FIGURE 8. UNIT GENERATOR.....19

FIGURE 9. AN ENVELOPE UNIT GENERATOR.....21

FIGURE 10. AN ENVELOPE GENERATOR DIRECTLY CONTROLLING THE AMPLITUDE OF ANOTHER UNIT GENERATOR.....21

FIGURE 11. TYPICAL SIGNAL FOR AN ADSR ENVELOPE GENERATOR.....22

FIGURE 12. MULTIPLICATION AND ADDITION UNIT GENERATORS.....22

FIGURE 13. EXAMPLE OF A MULTIPLIER USED TO CONTROL AMPLITUDE.....23

FIGURE 14. A DIGITAL OSCILLATOR.....24

FIGURE 15. ONE PERIOD OF A SINE WAVE.....25

FIGURE 16. THE UNIT GENERATOR SYMBOL FOR A FILTER.....27

FIGURE 17. AMPLITUDE MODULATION ILLUSTRATED.....28

FIGURE 18. PORTAUDIO PRODUCER-CONSUMER DIAGRAM.....53

FIGURE 19. A SUBSYNTH MODULE.....56

FIGURE 20. FRONT PANEL OF A HARDWARE MODULAR ANALOG SYNTHESIZER.....57

FIGURE 22. SUBSYNTH UNIT GENERATOR DESIGN [UML]. 60

FIGURE 23. AUDIO SYNTHESIS NETWORK AUTHORING TOOL. 62

FIGURE 24. THE PRODUCER CONSUMER PATTERN. 66

FIGURE 25. PRODUCER/CONSUMER IMPLEMENTATION IN SUBSYNTH [UML]. 67

FIGURE 26. AUDIO INPUT STREAMS [UML]. 69

FIGURE 27. AUDIO OUTPUT STREAMS [UML]. 70

FIGURE 28. SUBSYNTH MODULES [UML]. 73

FIGURE 29. SUBSYNTH BUILDER UML DIAGRAM. 75

FIGURE 30. UML DIAGRAM FOR THE INSTRUMENT TYPE. 76

FIGURE 31. SAMPLEBUFFER DATA TYPE. 77

FIGURE 32. SUBSYNTH TASK MANAGEMENT. 79

FIGURE 33. THE SCHEDULER ABSTRACT INTERFACE. 79

FIGURE 34. THE SUBSYNTHMIDI API. 80

FIGURE 35. L-SYSTEM STRING ELEMENT. 83

FIGURE 36. L-SYSTEM UML DIAGRAM. 84

FIGURE 37. UML DIAGRAM OF THE SONIX APPLICATION PROGRAMMING INTERFACE (API). 89

FIGURE 38. THE SONIX SOUND OBJECT SYSTEM [UML]. 91

FIGURE 39. SONIX PLUGINS. 92

FIGURE 40. TIME MEASUREMENT OF THE YIELD() FUNCTION. 95

LIST OF LISTINGS

Number

LISTING 1. A SIMPLIFIED UPDATE() PROCEDURE.....65

LISTING 2. EXAMPLE SUBSYNTHMIDI CONFIGURATION FILE.....81

LISTING 3. HOW TO RECONFIGURE SONIX AT RUNTIME 86

LISTING 4. CODE TO SETUP A SONIX SOUND.....87

LISTING 5. CALL SONIX::STEP() IN THE APPLICATION'S FRAME FUNCTION. 87

LISTING 6. EXAMPLE C++ PROGRAM THAT USES SONIX TO PLAY A SOUND USING SUBSYNTH. 88

LISTING 7. CODE TO STARTUP AND INITIALIZE SONIX TO USE THE SUBSYNTH AUDIO SUBSYSTEM. 89

LISTING 8. EXAMPLE OF AN INSTRUMENT DEFINED IN CSOUND'S INSTRUMENT DEFINITION SCRIPTING LANGUAGE.....100

LISTING 9. EXAMPLE OF A SUPERCOLLIDER SCRIPT.....101

LISTING 10. AN EXAMPLE OF A SUBSYNTH INSTRUMENT DEFINITION FILE.....102

LISTING 11. TEMPLATE METAPROGRAMMING AUDIO FORMAT CONVERSION ROUTINE.....104

LISTING 12. EXAMPLE MUSIC L-SYSTEM, USING THE L-SYSTEM XML SCHEMA SPECIFIED BY G.A.M.E.....106

ACKNOWLEDGEMENTS

I would like to thank my wife Laura for her patience with my long nights and weekends during grad school. I would like to thank my parents Mike and Susan for their support in my hobbies and education while I was a kid, and instilling an appreciation for science, music, and art, which drives me to this day. During my time at the Virtual Reality Applications Center from 1996 to 2002, I have met many people who have influenced me about art, music, public speaking, writing, good engineering philosophies and coding techniques. In addition there have been simply many friends and good times, many lunches, movies, and trips all over the world. For these I thank, in no order at all, Lew Hill, Allen Bierbaum, Ben Scott, John Walker, Benji Bryant, Herb Sawyer, Terry Welsh, Patrick Hartling, Justin Hare, Todd Furlong, Perry Miller, Matthew Steinke, and Laura Arns. For his help in implementing the initial prototype of Subsynth for a class project in 2001, I would like to thank Christopher Just. Lastly I want to thank Dr. Cruz-Neira for having me as her student; Iowa State University would have been quite different without her.

ABSTRACT

Today we find ourselves overwhelmed with information, and are faced with the challenge of how to input data into the human mind faster and more effectively. Visual display devices such as those found in television and computer displays are the most common method we use to get new information. One medium that is very effective, but not as common as visual presentation, is audio. Audio represents yet another channel of information that the human brain can easily perceive, and can even be processed in parallel to other channels such as visual, touch or smell. A subset of audio presentation methods that are useful is non-speech audio, such as sounds from music, sound effects, or warning tones. Non-speech audio may also be desirable when presenting many elements of data at one time. Examples of this can be found when simulating virtual environments such as three-dimensional games, or scientific simulations such as the visualization of genetic data. The audio needs for these kinds of applications can be met through the application of audio synthesis techniques. Audio synthesis is the field that investigates methods to generate and manipulate arbitrary sound in response to user input or data events. Until recently audio synthesis capabilities have been confined to specialized real-time hardware devices, or non real-time (offline) rendering on generic personal computers. In the last few years, personal computers capable of performing audio synthesis methods with real-time response are now affordable and commonplace. As personal computer technology advances, specialized real-time audio synthesis applications are becoming available. However, most of these applications are customized to specific software and hardware. These PCs we speak of run soft real-time operating systems such as Windows, Linux, Irix, and OSX. Real-time audio synthesis on these systems is possible, and several specialized implementations exist. One problem with custom solutions are limitations in scalability which affect how many simultaneous sounds can play, extensibility which affects how a user can add custom extensions to the synthesis tool, portability which affect what hardware the synthesis software can be run on, and generality which affect what types of synthesis methods one can choose. Alternatives to custom specialized

tools are emerging that allow synthesis of interactive and arbitrarily specified sound. We present a new audio synthesis toolkit, called "Subsynth", to facilitate building system independent and arbitrary audio synthesis components that are intended for use in real-time applications. To achieve this usability, Subsynth provides a cross platform and extensible software design. Subsynth runs in software on the same PC or workstation as the application that uses it. Applications use Subsynth through the C++ programming language. This thesis focuses on the approach taken to specify, design, and implement Subsynth as well as the benefits derived from this approach.

CHAPTER 1 INTRODUCTION

Research Problem

The sound processing power of computers has significantly increased in recent years to the level that most personal computers can store, reproduce, and transmit sound in digital format, i.e. *digital audio*. The process of generating and manipulating digital audio is usually referred to as *digital sound synthesis*. Synthesized sounds can be generated from existing sounds or from mathematical and physical models. Current digital audio capabilities in computers are enabling the possibility to create original sounds, mimicking existing instruments and voices, and creating sonic environments. These synthesized sounds are being used in many application domains; For example, virtual environments use synthesized sound to give users feedback about the state of virtual objects, or create environmental cues; scientific sonification makes use of sound synthesis to aurally display data; interactive music applies sound synthesis techniques to allow composers to dynamically control streams of audio events during a live performance.

There are a variety of existing sound synthesis tools available, both software and hardware based, varying in scope, implementation and cost. There are two approaches to these tools. One approach is with low-level toolkits, close to the hardware and operating system, which require additional programming to manipulate the sound. Another approach is end-user applications, generally built from the low-level toolkits, which are accessed with a graphical user interface (GUI).

Audio synthesis toolkits are often implemented in specialized audio hardware due to their computational needs. Generally, most tools implemented in hardware cannot migrate or scale to other systems and configurations beyond those of their original design, limiting their use by a wide range of users. Lately personal computers have become fast enough to support software-based audio synthesis, which potentially enables the audio generation to happen on a heterogeneous mix of computing platforms, including the ability to run on the same system as the end-user application.

Software audio synthesis has the potential to provide the features of scalability, portability, and generality to end-user applications. When audio synthesis tools are not scalable, they limit the growth of audio capabilities for new applications. When these tools are not general, then it is possible they will not fit the task, forcing a developer to use combinations of audio tools or writing new ones. When using tools that are not portable, applications that are otherwise portable cannot migrate to other systems. This thesis addresses those issues by presenting an audio synthesis toolkit design for the creation of interactive audio applications that can be portable, scalable, and reusable across heterogeneous underlying audio hardware capabilities and computer systems.

Statement of Purpose

The research presented in this document addresses the generality, scalability, and portability problems associated with real time audio synthesis toolkits used by end-user interactive applications. The research focuses on the design and development of Subsynth, a low-level audio synthesis toolkit for building higher-level real-time audio components for use in virtual environments, scientific sonification, and interactive music composition and performance.

Scalability in Subsynth is achieved by decoupling the sound generation from the hardware, allowing applications to transparently increase their audio capabilities as more powerful audio hardware becomes available. Generality in Subsynth is achieved by using a modular design, low coupling of components, and parameterization of configuration information, enabling reusability of audio synthesis components. Portability is achieved by Subsynth's hardware and operating system independent implementation. In addition Subsynth is distributed as an Open Source product, so hopefully a large community of users will benefit from using it, and audio developers will be able to enhance it and evolve it as new technology in audio synthesis becomes available.

Subsynth is intended to be a flexible layer to support the development of higher-level audio tools for interactive audio applications. The goal is not an interface to audio for laypeople but rather a scalable, portable, and open audio subsystem architecture, built around the concepts of audio synthesis, to make possible a wide range of audio applications.

Scope of Research

The scope of this work applies to interactive audio for virtual environments, scientific sonification, interactive music, and general audio playback.

The research was performed in the following stages:

1. **Define requirements specific to real-time audio applications**

   We first needed to analyze the application domains of virtual environments, interactive audio, and scientific sonification; then we identified their audio requirements. To help understand the requirements, existing tools were examined, and some prototypes were developed.

2. **Analyze existing audio subsystems**

   From the requirements addressed in the previous step, we investigated how existing tools addressed them. We noted strengths and limits, and used these to guide our design.

3. **Design of Subsynth based upon requirements**

   By analyzing existing audio tools in step two, and combining that with the requirements found in step one, we were able to define a design for Subsynth. Iterating between steps one, two, three and four yielded a process of iterative refinement used to finalize the design of Subsynth.

4. **Implementation**

   Once we had a design for Subsynth we implemented the core framework, which includes audio generators and processors, and the methods in which to configure them generically. A stable core allowed multiple developers to work on separate components at the same time. Using the implemented core framework, we created extensions to enhance the functionality of the toolkit. We also developed libraries and applications on top of the overall tool. This usage of the framework

helped us to see how well the design of the framework held up, and it caused us to look for new ways to improve the framework design.

5. Develop case studies

With a stable core and a set of useful features in place, we were able to write applications to exploit the benefits of the Subsynth software. To test the real-time and functional capabilities of Subsynth, we used its audio synthesis capabilities in the context of generating music. Also, we noticed that one of our early prototype sound toolkit designs (called Sonix) could be implemented in terms of Subsynth. Sonix turned out to be much too simple for the generalized requirements of Subsynth but was useful to a very common computer audio use. With this new tool Sonix, we can provide a high level and simple interface on top of Subsynth that enables a novice programmer to use audio quickly.

6. Discuss Results and Future Work

With the research completed, we are able to reflect upon the successes and the failures. This is the last iteration before the completed research, marking the current state of Subsynth. Sharing of results enables other researchers to understand the effectiveness of the research and communicates to future researchers where to go next.

The research stages are presented in this document as follows:

- Chapter 2 covers background material and definitions for Digital Audio, Computer Music, and Sonification.
- Chapter 3 outlines the needs of an interactive audio tool.
- Chapter 4 illustrates stage two of the research by presenting an overview and analysis of existing interactive audio tools.
- Chapter 5 satisfies stages three and four of the research by describing the analysis, design, and implementation of Subsynth.
- Chapter 6 shows stage five of the research by showing case studies where higher level tools have been built upon Subsynth.
- Chapters 7, 8, and 9 satisfy stage six of the research by discussing the results and future work needed for Subsynth.

CHAPTER 2 BACKGROUND

Digital Audio

Most computers today have audio capabilities that range from producing simple sounds triggered by user interactions to the low level rendering of high quality music. These audio capabilities are possible through the digital representation of sound. In the real world, sound travels as longitudinal waves compressing and expanding the air. If we imagine a molecule of air being pushed by the waves, we can represent the sound by a function of time $x(t)$, where x is the displacement of the molecule (air pressure). To represent sound in a computer, the time variable t and the pressure variable $x(t)$ must be discretized by two processes called *sampling* and *quantizing* [Dodge97][Steiglitz96].

Sampling is the process of taking measurements over time resulting in a discrete series of values. Quantizing is the process of clamping each measurement to one of a discrete range of values. Usually each measurement is called a *sample*, while the entire waveform is called the *signal.* Samples can be obtained by taking individual measurements from sound signals. In addition, samples can be generated procedurally, which is common in sound synthesis, and they can be processed using manipulative functions such as distortion or filtering, which will be explained later. Understanding this process of sampling and quantization is important to this thesis work. Because Subsynth is an audio synthesizer, it will need to generate and process samples over time. The audio synthesis algorithms that do the generation and processing need to be aware of the sampling rate, and the resolution at which pressure values are quantized.

Sampling

Sampling is the process of taking pressure measurements from a signal over time. Sampling theory tells us that when representing a signal digitally, there must be two times the number of

samples per second than the highest frequency expected to reproduce [Steiglitz96]. This highest possible frequency is called the Nyquist frequency. It represents the highest reproducible frequency possible when using a given sampling rate. Nyquist tells us that, for a given sampling rate, the maximum frequency that is representable is always one half the sampling rate. Therefore, the sampling rate always needs to be more than twice as high as the highest frequency that exists in the sampled signal. To understand this, imagine sampling a steady single-frequency signal at a lower rate than the signal's original frequency (Figure 1). The result is that the sampled data would be missing much detail since we are trying to capture frequencies well above the Nyquist frequency, which would be around 350Hz in the figure.

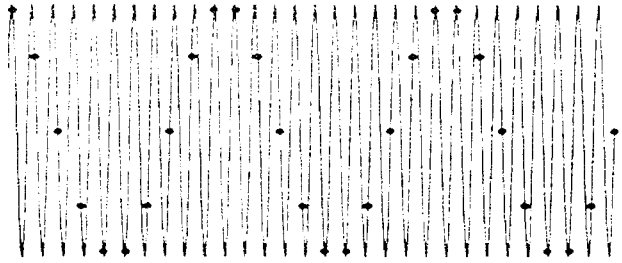

Figure 1. Sampling a 770Hz sinusoid at 700Hz.

In Subsynth, the sampling rate needs to be considered so that audio signals are generated and processed at the same rates. The Nyquist frequency is very important since it determines the highest possible frequency tone that a synthesizer can generate. To generate any sound, a synthesizer needs to sample existing signals, whether they are from the real world, from pre-recorded sounds, or from procedural functions. In pre-recorded sounds, even though represented digitally, they can be sampled at higher or lower rates to achieve a change in frequency. In procedural functions, for example $f(t)$, they need an increment in t (time) each time the synthesizer needs to get a new value. Each call to $f(t)$ is considered one sample.

Quantizing

Quantizing is the process of representing a signal's pressure values with some finite bit-length word. Computers today are able to represent a signal with no noticeable artifacts due to word length. It has become practical to use 32 bits or more per sample. Artifacts from smaller word lengths occur because of error. For example, if a pressure value at a given time sample is 12349.5, but during quantization the value is clamped to a number in the set of integers, then this error and many others in the signal may become noticeable when listening to the signal as a whole.

To visualize this phenomenon of quantization error that we have just explained, see Figure 2. It is important to notice that whenever the computer samples a signal, whether originally from the natural world or from some mathematical function, the resulting digital signal will have distortions proportional to the word length chosen to represent the signal. To enhance sound clarity in an audio synthesizer such as Subsynth, a large word length should be chosen, but not so large that performance is impacted. Larger sizes mean that more data will have to be processed, although some processors have specialized units available that can make processing of larger sizes actually faster than using smaller sizes. Currently 32-bit floating-point values work well on modern processors.

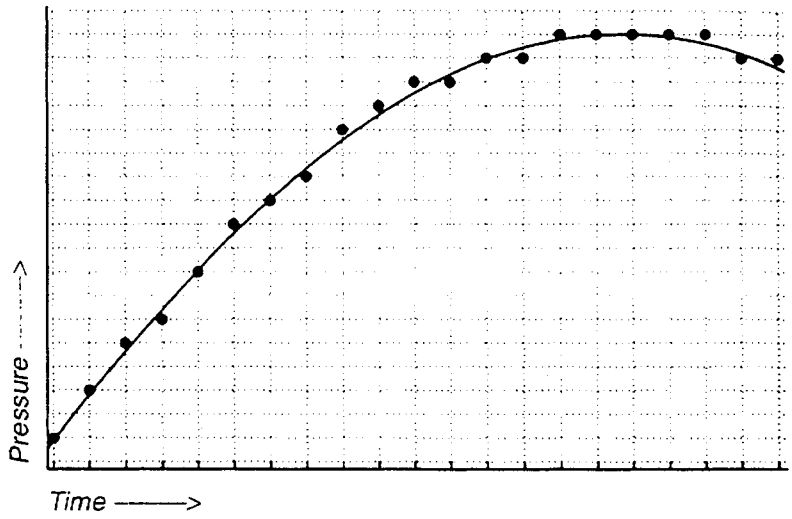

Figure 2. Errors during quantization of a smooth curve.

Digital/Analog Conversion

To further understand the relationship between real audio signals and digital ones, we discuss a process to convert between them. In the real world, audio signals exist as an infinite series of pressure values, while in a computer, they are represented by a finite series of discrete pressure values. A conversion process is used to go between the real world and the computer representations of any audio signal. Sampling and quantizing achieve this conversion process. Conversion is done by a piece of hardware called the analog to digital converter (ADC), which is usually a small integrated circuit (IC) computer chip. To sample, the chip measures the incoming sound pressure value at regular intervals called the *sampling rate*. To quantize, each sampled value is then clamped to one of a discrete range of values (Figure 2).

When reproducing sound, the system needs to convert the digital signal back to an analog format suitable for speakers, headphones, or other analog equipment. This reverse conversion process is done with a similar IC called a digital-to-analog converter (DAC). Conversion from continuous to discrete and vice versa is important to Subsynth to help understand the relationship between natural audio signals and those being generated and manipulated.

Dual Representation

Computers represent sound in terms of digital signals, but there are two ways to represent them. A digital signal can be represented as a *waveform* (pressure versus time) (Figure 3) or as a *spectrum* (frequency versus time) (Figure 4). Pressure vs. time is very common and is the usual method of storing audio signals, however either representation is useful in different situations. The waveform representation is useful when manipulating aspects of time, while the spectrum representation is useful when manipulating aspects of frequency.

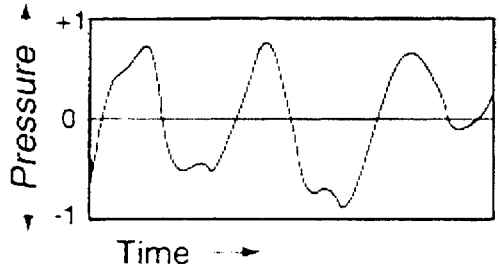

Figure 3 Time domain representation of a signal.

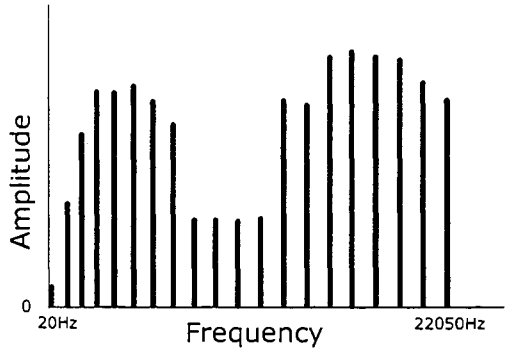

Figure 4. Frequency domain representation of a signal.

Each representation may be converted to the other using a Fourier transform. There is an efficient version of this algorithm suitable for real-time use called the Fast Fourier Transform (FFT) [Steiglitz96]. Conversion is appropriate when manipulations in one domain are easier to think of or more efficient to compute than in the other domain. The waveform is constructed from the spectrum by generating sinusoids over time with the list of frequencies.

These two representations are important to a digital synthesizer that needs to perform operations in each time or frequency domain. For example, to change the pitch or frequency of a tone without affecting the duration, the synthesizer will use an algorithm that changes the frequency domain attributes of the waveform. To add echo effects, the synthesizer should use the time domain representation since an echo is dependent on the sound propagation time through the air. There are many effects possible when considering this representation duality.

Distortion

One method of manipulating an audio signal is distortion. Distortion can be qualitatively "good", for example in the case of vibrato, which is a method of modulating the frequency of a tone slightly over time. Distortion can also qualitatively be "bad", in the case when unwanted perturbations happen to a signal, which we will describe next. It is usually important to minimize distortion in a digital audio system to minimize perceivable artifacts. There are three types of unwanted distortion: frequency, amplitude, and phase distortion [Steiglitz96].

Frequency distortion is caused by errors in an electronic device, either through faulty electronics, or digital effects such as round-off and precision errors. For example, frequency distortion could occur when specifying a given frequency, and the synthesizer creates a tone with a frequency that is off by some amount. For example, this could happen when converting a control signal into a frequency where the conversion process is faulty or uses approximations. Code optimizations are sometimes used which choose speed over accuracy. Control signals are usually represented linearly for intuitive use by humans, while frequency is on an exponential scale, as illustrated by the fact that a given pitch doubled in frequency, sounds like the same pitch an octave higher to the human ear.

Amplitude distortion is the most common, and is caused by the non-linear response of a device to the input signal amplitude. Non-linearities occur, for example, when boosting a signal above the maximum limits that the hardware or numerical representation allows. This is often called *clipping* the signal.

Phase distortion can occur when there is some latency in a particular spectrum. This can occur in hardware when using separate speakers for bass and treble. Alternatively it can occur with a digital system that processes different components of the spectrum with different latencies.

In a synthesizer, distortion should be avoided except where explicitly needed during processing. For example, when reproducing a tone verbatim, care should be taken to avoid quantizing the sound pressure values outside the possible ranges that the datatype will hold. This is an example of *clipping*, which is distortion that is usually undesired. For an example of distortion that is desired, see the section later on modulation synthesis. Modulation synthesis can distort one signal using another signal to achieve effects such as tremolo and vibrato.

Common Audio Formats Used in Digital Computers Today

This section covers the practical issues of implementing the digital audio representation inside a computer. It gives several common data formats that are used in synthesis applications and forms a practical guide for how to represent digital audio in the Subsynth synthesizer.

An audio format is a specification of how to represent one or more audio signals in computer memory, some examples of which could be hard disk, RAM, optical disk, or digital audiotape. These media types have uses ranging from long-term audio storage for archival to short term storage for immediate playback. Some formats require more processing than others to retrieve. Each format has strengths and limitations, and here we will discuss in detail PCM, WAV, AIFF, MP3, and OGG, the most popular audio formats.

These formats range from open public standards to closed proprietary formats. MP3, which is a closed format, is not open but is in wide use and generally perceived to be open. Closed formats such as MP3 can have patents, which may be claimed at a later date. When choosing to use one of these formats, it is good to consider these issues in addition to the technical merits.

PCM

Pulse Code Modulation, or PCM, is the most common format for storing uncompressed digital audio [Steiglitz96][Roads96]. It is the format found on compact discs and is the intermediate format

used to transmit audio data to a computer's sound card during playback of any audio file whether during game play, virtual environment usage, or simple playback of a music album. The PCM audio format consists of an array of sound pressure levels sampled over time (Figure 5). The number of samples per second is the sampling rate, and the bit width of each sample affects the quantization. There can be many of these arrays to represent sound channels. Sound channels are used, for example, in sending discrete information to individual speakers, or to other audio components for further processing. Each sound channel may be interlaced (Figure 7) or separated (Figure 6). Interlaced PCM audio format is the most common format because each sample in a given frame is readily available as a result of their close grouping in memory.

Figure 5. Single audio channel of PCM data in memory.

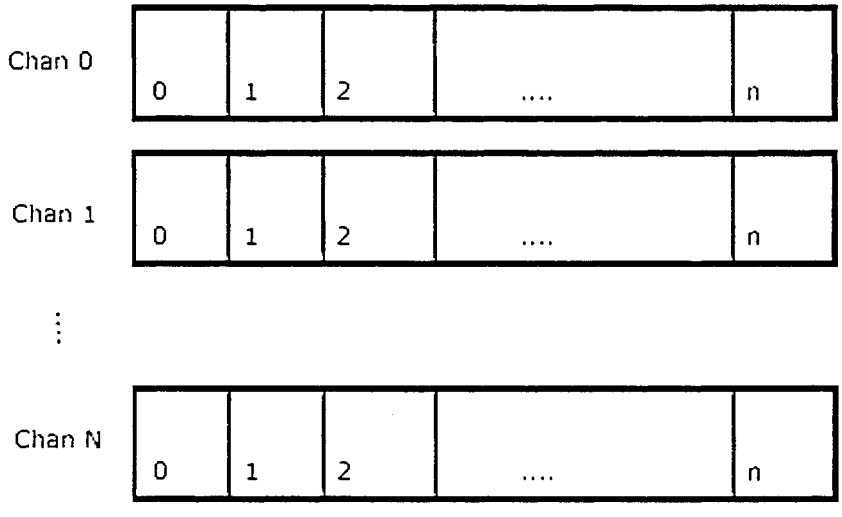

Figure 6. Noninterlaced PCM audio data with N number of channels.

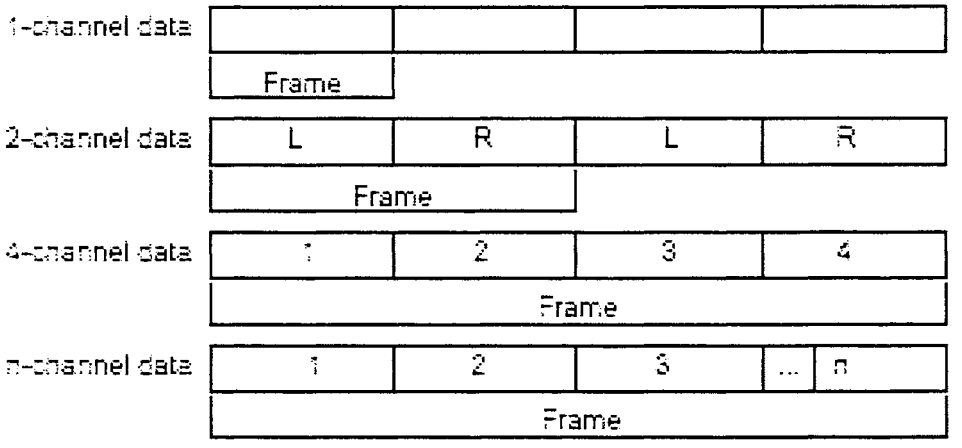

Figure 7. Interlaced PCM audio data.

PCM is a fundamental audio data format. In practice, though, PCM has its strengths and weaknesses. Strengths include quality and lossless representation of an audio signal. Weaknesses in PCM are in its size, which can make distribution of recorded material more expensive. There is much more data in PCM than the human brain can process. For example, each minute of 44100Hz 16bit stereo PCM data takes about 10 megabytes of storage. In Subsynth, and in many other software synthesizers such as the ones we review in this thesis, audio is processed in the fundamental PCM format.

WAV PCM

The WAVE file format, or WAV, is a format for storing digital audio data [WavSpec]. It supports many different combinations of encoding types, bit resolutions, sample rates, and channels. The most simple and common encoding used in the WAV file format is the uncompressed PCM format mentioned earlier, or WAV PCM. WAV also supports a variety of other encoding methods, making it a sort of "catch all". WAV also stores some additional meta data for *cue points*, *play lists*, and *instrument* data. A cue point specifies a marker to a specific offset within the data. A play list specifies a play order for a series of cue points. The instrument definition is used to specify

information useful to wave table synthesis such as gain, tuning, and loop points. The WAV format is very common on computers running the Microsoft Windows operating systems.

The strength of the WAV PCM format is that it can very closely match the original recording, especially if the PCM data was taken verbatim from a compact disc recording. The weakness of WAV PCM is in its size, where CD quality stereo PCM data is about 10 megabytes per minute. With the instrument data field definable, the WAV format would be good for audio synthesis applications.

In Subsynth, WAV PCM is used to read and write sounds from the computer's file system. The following formats we describe in this section (AIFF, MP3, OGG) could be used but currently are not supported, we explain them because they would be useful to have in a synthesizer such as Subsynth.

AIFF

Audio Interchange File Format, or AIFF, is another format similar to the WAV format [AiffSpec]. It also supports PCM encoding, as well as other meta data to describe instrument parameters such as loop points and MIDI note value. AIFF format is primarily found on the Apple operating systems, but is also common on some variants of Unix such as SGI's IRIX. With respect to PCM encoded data, the strengths and weaknesses of AIFF are roughly the same as the WAV format.

MPEG Layer 3

MPEG Layer 3, or MP3, is an audio format that allows a high compression ratio while maintaining sonic quality [MP3Fraunhofer][MP3Spec]. MP3 format uses a perceptual encoding technique based on knowledge of human psychoacoustics to achieve compression up to eleven times smaller than the original uncompressed audio signal.

Encoding of MP3 data removes both absolute and perceptually redundant information and is performed using two compression techniques, one which is lossy[1] and based on mathematical models of human psychoacoustics, and one which is lossless[2]. First, the source audio data is transformed to its frequency domain equivalent. Then, for each frame, the frequency spectrum energy distribution is analyzed and compared to models of human hearing. Based on human psychoacoustics, certain frequency components can be discarded. Some cases where certain frequencies are not needed are:

- When two frequencies are very close, the human brain usually only hears one of them.
- When one frequency is much louder than another, the human brain may not hear the quiet one.
- When frequencies are outside the range of normal human hearing, they will not be heard.

After unneeded frequencies are culled out using lossy compression, then a lossless compression technique called "Huffman Encoding" is applied. This is a standard compression algorithm, similar to the common ZIP compression, which removes any redundant data left in the audio stream [Murray96].

The data in an MP3 file is broken into frames of a few milliseconds in length, each frame with a 32bit header preceding it that describes the bit rate, sampling rate, mpeg version, a data checksum, and other bits that describe the file such as whether it is copyrighted or an original file. The data part in each frame is encoded by frequency, which during playback, can be converted to the time domain form.

---

[1] Lossy refers to a data compression technique that actually reduces the amount of information in the data, rather than just the number of bits used to represent that information. This technique's approach is based on the assumption that the removed information is not important for the quality of the data, which is usually an image or a sound.

[2] Lossless refers to a data compression technique that retains all the information in the data, allowing it to be fully recovered by decompression.

Strengths of MP3 include its very small size in relation to PCM. For CD quality stereo the general rule is about one megabyte per minute. This is very important for small-memory embedded devices, storage of large audio databases, and for transmission of audio over slow network connections. The perceivable quality of MP3 is very good, where most people cannot tell the difference between MP3 and PCM. One very useful application of this technology is in collaborative virtual environments, where upon connection, a client may need to download or refresh their copy of the world database. MP3 technology, with its very small memory footprint, brings audio in collaborative virtual environments to a wide audience of users who may have very slow connections.

Weaknesses of MP3 are that the format is lossy, and while it usually cannot be perceived, the lossiness is perceivable by certain people with sensitive hearing. Also, artifacts from the lossy compression will build up over time as the same piece of audio is repeatedly compressed and decompressed. One other point to mention about MP3 is that it is a proprietary format whose patent is owned by Fraunhofer. This means that, technically, any use of MP3 must pay royalties to Fraunhofer.

OGG Vorbis

OGG Vorbis, or "OGG", is a format similar to MP3, but developed to be completely patent free. It offers an open standard freely available to anyone. It has all the strengths of MP3, but with better quality for smaller file sizes [Ogg]. OGG, like MP3, is also lossy but is not restricted by patents. For open source and public domain applications where data size is a concern, OGG would be a good solution. Synthesis applications should be aware of the processing needed to convert OGG and MP3 to the PCM format, which is useful to audio synthesis algorithms. OGG is not supported yet in Subsynth, but should be in the future in order to facilitate high compression for the fast distribution of data.

Digital Audio Synthesis

Digital audio synthesis is a way to generate the time-domain sequence of numbers that represents the samples of an audio waveform. In short, sound synthesis is a way to generate sound. There are many ways to synthesize sound, each allowing a different method of control. Audio synthesis capabilities are important to Subsynth giving us methods to produce audio signals than can be modified in real time. The audio synthesis methods we will examine are as follows:

- Table lookup synthesis generates waveforms by sampling a stored function representing a single cycle.
- Additive synthesis generates the individual frequencies of a complex tone, each with its own loudness curve (or envelope).
- Subtractive synthesis begins with a complex tone and filters it.
- Non-linear synthesis uses frequency modulation and wave-shaping to give simple signals complex characteristics.

To specify these forms of audio synthesis, and combinations of them, a visual modeling language is available called the "unit generator language". This language translates into the basic framework for Subsynth that provides a method to generically hook together audio generation and processing modules. Next we describe this language, and then we use this language as a basis to describe several audio synthesis methods.

Unit Generator Language

When the Unit Generator visual language was developed, it was considered a very significant development in audio synthesis language development. Many kinds of audio synthesis algorithms are specifiable using the unit generator language. The first audio synthesis language using the unit generator concept was Music III by Max V. Mathews and Joan Miller in 1960 [Roads96].

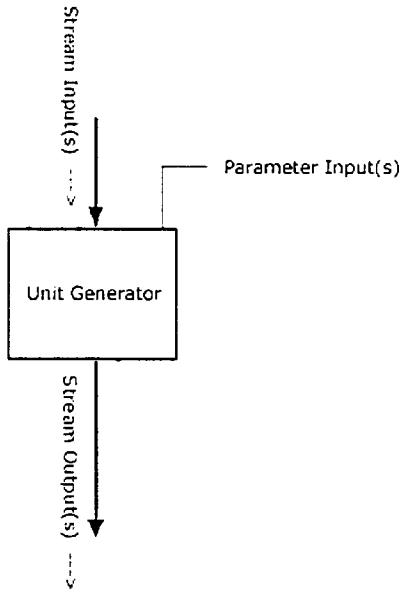

Figure 8. Unit generator.

Unit generators are connectable signal processing modules that take as input zero or more audio signal streams or single valued parameters [Roads96][Dodge97][Moore98]. Shown in Figure 8, unit generators consume real-time and parameterized input, and produce real-time output signals that can be connected to other unit generators. The information carried in these real-time signals can be audible sound or for high-resolution control. After processing, a signal is output at zero or more data streaming outputs. A sink, which is a unit generator with only inputs, is used to represent a processor with no output onto the defined sound network. Sinks are often used to represent transport of the audio data to the outside of the system, for example to a sound card or other subsystem external to the synthesis software. Often each source unit generator is called a "voice". When connected, the network of unit generators form an instrument or "patch" that can generate a sound signal. Many commercial synthesizers use the concept of patch to represent a single instrument, or more specifically, a predefined configuration of a unit generator network. In practice, the patch may or may not be implemented as flexibly and as modularly as the unit generator language. The concept is

an effective way to think about what is happening in the synthesis algorithm regardless of how it is implemented.

Unit Generator Building Blocks

Before we introduce some audio synthesis methods, the anatomy of unit generators, and several unit generator types should be understood first. These types are used to describe the various synthesis methods:

1. **Envelope and Envelope Generator.** A constant value used in the amplitude of a waveform yields a sound event with constant amplitude. In the real world, instruments and other sound events often have time varying amplitude, and in general sound more interesting than non amplitude-varying sounds. To support time varying parameter input, the concept of *envelope* is introduced. An envelope generates a signal that is used to control other unit generator's parameters, such as amplitude or pitch, over time. Envelopes can be specified in many ways. In Figure 9 an envelope generator is shown that uses a waveform to control amplitude. In practice, many commercial synthesizers support a different style of envelope called an *ADSR*, which lets the user specify the time delay and amplitude of the attack, decay, sustain, and release events (Figure 11). An ADSR allows a simple, efficient, yet course-grained method of control of amplitude over time. Some current digital synthesizers allow tracing of arbitrary curves, which allow greater precision in specifying the behavior of an instrument. Envelope generators may be implemented to generate their output via a procedure (mathematically), or from a specified waveform.

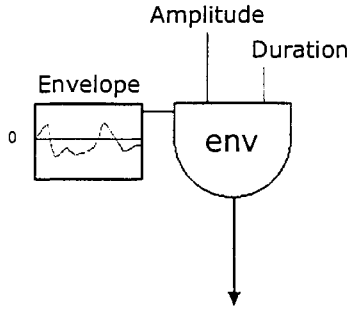

Figure 9. An envelope unit generator.

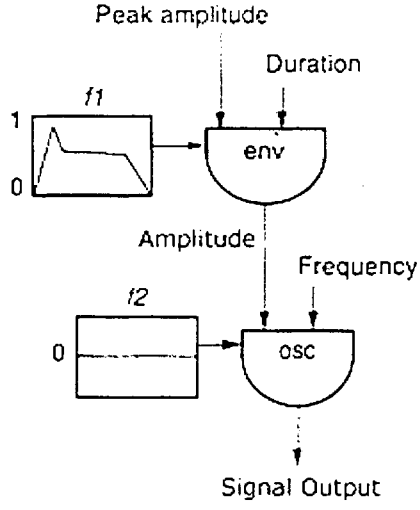

Figure 10. An envelope generator directly controlling the amplitude of another unit generator.

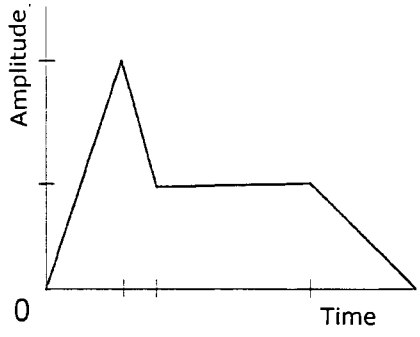

Figure 11. Typical signal for an ADSR envelope generator.

2. **Arithmetic Operator**. In audio synthesis it is often necessary to combine or transform signals with other signals. For this purpose an arithmetic unit generator is useful. Common

operations include multiplication and addition (Figure 12), which are used to combine signals together. Multiplication is useful when scaling the amplitude of a signal. For example, to apply an envelope to an existing signal, attach the envelope generator to one terminal of a `multiply` unit generator, and attach the signal to the other terminal (Figure 13). Addition is useful when mixing two or more signals.

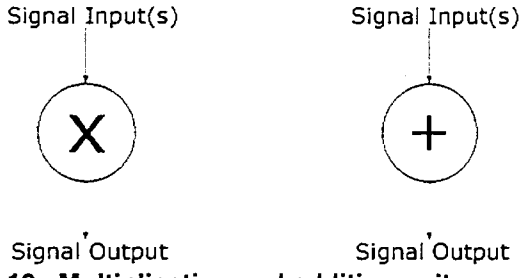

Figure 12. Multiplication and addition unit generators.

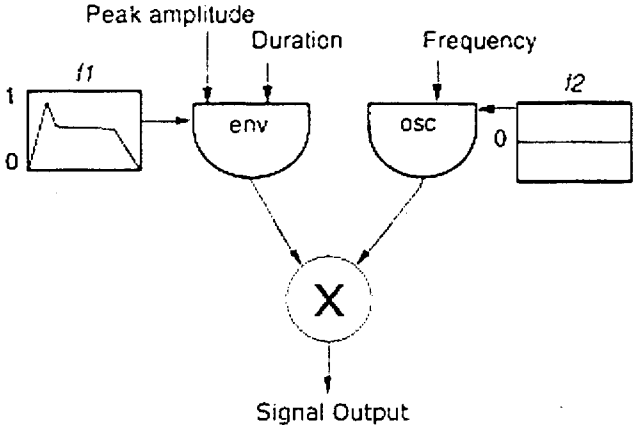

Figure 13. Example of a multiplier used to control amplitude.

Fixed-Waveform Table-lookup Synthesis

Fixed-Waveform Table-lookup Synthesis, or wave-table synthesis, is an efficient way to produce complex tones [Dodge97]. Wavetable synthesis takes advantage of the fact that some sounds contain redundant information. At small scale, redundancy comes from the simple periodicity of frequency,

at larger scales redundancy comes from sounds that contain loopable components – parts that when repeated make it hard to hear a repetition.

Wavetable synthesis works by looping a portion of the sound that is repeatable – or with the ability to be repeated so to not become tiring to the ear. A wavetable is analogous to textures from the field of computer graphics. Often a texture will be used as an easy and effective way to add complexity and detail to 3D surface geometry. Textures should be tileable, or able to be repeated, without the ability to discern where the tiles start or stop. Like textures, wavetable sounds also need to be tileable, except that the words loopable and repeatable apply better to the case of sound. Representing complex features using presampled material is advantageous. The reason is that in terms of computer processing power, reading samples from memory is much less expensive than mathematically computing each sample.

The Digital Oscillator Unit Generator

A digital oscillator is a concept that describes the traversal over an array of data. The array can be circular, meaning that the traversal can iterate through the data restarting at the beginning causing a loop, or the array can be noncircular where the traversal can travel to the end once and simply stop. A digital oscillator is a fundamental type in the unit generator language (Figure 14). This unit generator is commonly used in synthesizers because of its efficiency and realism.

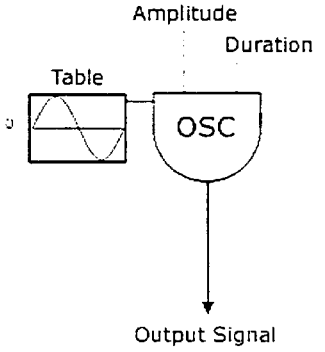

Figure 14. A digital oscillator.

Example Uses of Wavetable Synthesis

Wavetable synthesis is very common in modern synthesizers because of its efficiency and realism, making the price per performance very attractive. Practically any real instrument can be synthesized using this technique. To create a loopable sound for use in a wavetable, the instrument is sampled within the frequency range expected to be used. Next the sound is edited to be loopable, which can be done by visually inspecting the waveform graph and alternately listening for artifacts.

Instruments that have a long period of sustain are especially good source material to use to obtain a repeatable sound for use in a wavetable. Usually post processing is required to make the looping wave indistinguishable from a real instrument. Alternately percussion instruments often do not repeat at all. To create source material for these, the duration of the instrument's play time is stored in a waveform. Post editing for repeatability is not necessary for these types of percussive instruments. Very simple sounds can also be represented with a wavetable. A sine wave is one example that has very short repeatable sections. All fundamental waves (sine, triangle, saw, square, etc.) by their periodic nature can be represented by a wavetable that is equal in length to one period of the waveform (Figure 15).

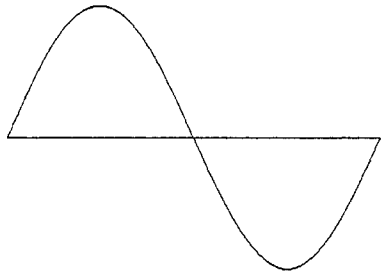

Figure 15. One period of a sine wave.

Implementing Wavetable Synthesis

To create the wave table entries, a signal can be loaded from file (see section titled "Common Audio Formats Used in Digital Computers Today"), procedurally generated using other synthesis

method described here, or sampled from the environment. Wave-table synthesis is implemented by traversing the wavetable values one or many times. Non-looping percussive instruments are traversed once, while other instruments with a long sustaining portion are usually traversed indefinately. Traversal of the values can be one to one, but it is often useful to support a variable step size, which allows the traversal speed, and ultimately the sound frequency, to be changed. Changing the traversal rate causes values to be skipped or selected multiple times. A better way to think about this traversal is that the digital oscillator is actually sampling the lookup table as if it were a continuous signal. Obviously, it is not a continuous signal, so when a sample falls outside of a real measurement some kind of approximation is given either by clamping to the nearest value or interpolating between the two nearest. To avoid high frequency artifacts caused by rounding errors during sampling, interpolation schemes can be used when the sample falls between two wave table values. Interpolation, rather than simply rounding to one value or the other, smoothes out the result, resulting in less artifacts and a waveform that is closer to the original in tonal quality. Several forms of interpolation are available: linear, cubic, and spline are a few (Interp).

Additive Synthesis and Wavestacking

Additive synthesis [Mathews69] is a method for creating complex sounds through addition of simple tones [Roads96]. For the simple tones, known as *partials*, additive synthesis uses sine waves. The addition of the partials yields one additive synthesis sound generator, known as a *voice*. A variation on additive synthesis is called "wavestacking", which adds together complex waveforms such as sampled sounds. Like additive synthesis, each sampled sound in wavestacking also has its own envelope. An example use of this could be to make hybrid instruments such as piano/flute, where the individual voices fade in and out according to their own amplitude envelope. An instrument using additive synthesis can produce a very wide range of sounds. The tradeoff with this

approach is that the number of input parameters needed to specify an additive synthesis algorithm grows with the complexity of the algorithm, making it very difficult to handle.

Subtractive Synthesis

Subtractive synthesis is the method of selectively removing frequencies from a source signal to shape the spectrum of a sound [Pierce92]. The dominant unit generator in subtractive synthesis is the filter. The filter takes as input one audio signal, and outputs one audio signal. Controls are usually provided so the user can affect which frequencies get removed. A common control is cutoff, which specifies the point at which to eliminate all frequencies either above or below depending on the filter. Typical filters are: the high-pass, which removes frequencies below the cutoff point; low-pass, which removes frequencies above the cutoff point; and band-pass, which is a combination of low and high-pass filters, which offers two cutoff points and results in only a range of frequencies being let through. Figure 16 shows the unit generator symbol for a band-pass filter, which only allows a range of frequencies through. This filter can also be configured to be a low or high-pass filter by setting the center at 0 or at Nyquist and using the BandWidth input as the cutoff.

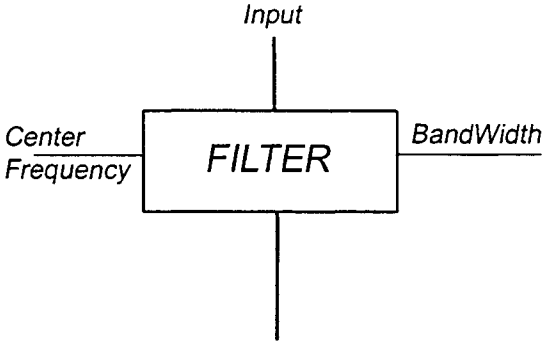

Figure 16. The unit generator symbol for a filter.

Non-linear or Modulation Synthesis

Modulation synthesis is a way to produce complex sounds with very simple groups of unit generators [Chowning73]. It includes several techniques, such as ring modulation, amplitude modulation, and frequency modulation. Specifically, modulation is the alteration of the amplitude, phase, or frequency of an oscillator as controlled by another signal [Dodge97]. Modulation synthesis is more efficient than subtractive and additive synthesis in terms of memory, processing time, and number of parameters to specify. Only 2-6 unit generators are typically needed for this type of audio synthesis, where several times this amount would be needed to achieve the same result in subtractive or additive synthesis methods [Roads96]. The shortcoming of modulation synthesis is that it can be very non intuitive to specify the parameters, since small changes in parameters yield widely different complex sound. With this type of audio synthesis it is possible to produce tones that come very close to natural instrument tones, as well as other more creative non-natural tones.

The configuration seen in Figure 13 is one example of how to connect unit generators for amplitude modulation (AM), or more generally, ring modulation (RM). RM is where one signal scales the other signal over some time varying frequency. The difference between AM and RM is that in AM the waveform is always *unipolar* (the entire waveform is above zero). At slow rates RM produces the popular *tremolo* effect, and at fast rates can produce more complex effects. Another possible configuration for RM is in Figure 10. Frequency modulation can be configured similarly to Figure 10 if the first oscillator was connected to the frequency terminal instead of the amplitude terminal. AM is also used to apply an envelope to a signal, a technique that produces a more physical quality for use in sounding musical notes. In Figure 17 we illustrate how an envelope can be used to shape a signal. The operation shown is c = a * b, where signal b scales signal a to produce signal c. In practice, each sample a[n] and b[n] are multiplied together to produce c[n], where n is a sample index.

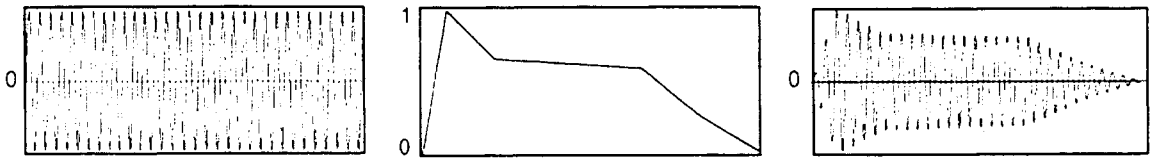

Figure 17. Amplitude modulation illustrated.

In summary, digital sound synthesis builds upon digital audio by specifying methods to manipulate the audio. Next we will introduce sonification, which is a discipline that uses the concepts we presented here in the Digital Audio Synthesis section. In sonification, audio synthesis is the technique used to manipulate audio in response to data in many situations.

Sonification

Sonification is a non-speech aural way to represent information [Kramer97]. It is a way to transform a given set of data into sound in order to better communicate or interpret the information. There are other terms to describe this idea, but "sonification" is the most common. The terminology is not standardized, but often the distinction is made between data-controlled sound (sonification), and direct playback of data samples (audification) [Kramer94]. Another term is "auralization", which is sometimes used as the sound parallel to "visualization", which is the standard term used to describe the representation of data through images [Visualization].

Sonification is an interdisciplinary field requiring skills from many fields of study including human perception, acoustics, design, the arts, and engineering. Sonification integrates the following four areas:

- Psychological research in perception and cognition
- Design and application of sonification methods
- Development of sonification tools

- End-user sonification applications

Motivation for Sonification

As computing power and storage capabilities increase, it is possible to study more complex scientific problems that generate or require massive amounts of data. For example, the Human Genome Project has generated huge amounts of data while mapping the human genome that are well beyond terascale needs. There are 30,000 genes in human DNA, with a total of 3 billion chemical base pairs [Genome]. There is a greater problem of how to analyze the data, or preprocess it to be ready for use with any display medium, let alone sound. But once that is ready, it is anticipated that the data will still be quite large, and in order to be analyzed by scientists, it will require several representation channels besides visualization, where sonification will probably play a key role.

Examples of Sonification

Sonification has been used successfully in many applications. Here we will present an overview of some of the greater successes. Simple uses of sonification include:

- **The Geiger counter**, invented by Hans Geiger in the 1900's, is a simple device to detect invisible radiation levels through "click" sounds [Kramer94]. Each click corresponds to a particle hitting the sensor allowing the user to perceive the level of radiation near them without taking their eyes off their work. The counter provides a continuous reminder of the current radiation level.

- **Sonar** is a method of detecting objects under water. Similar to radar, sonar sends pulses of sound through the water that reflect back. The time that it takes to get the signal back indicates the distance of objects, whereas the intensity of the sound tells the size.

- **Cockpit and dashboard displays** often use sound to let the pilot focus on navigation. For example warning tones emit when fuel is low so that the driver does not need to remove their eyes from the navigation task.

- **The Pulse-Oximeter** is an instrument that emits a tone that varies in pitch with the level of oxygen in the patient's blood [Kramer97]. This allows the surgeon to keep their eyes focused on the patient. This sonification of the oxygen level uses a different channel of information, freeing the surgeon to focus on what is important.

- **In seismology** recorded datasets can be huge and hard to understand [Kramer94]. Since seismic data is naturally acoustic, it makes sense to listen to it. Because the periodic frequency in seismic waves are much lower than the human audible range (20Hz – 20kHz), to hear them they need to be played back at a faster rate. A 24-hour dataset can be compressed to 4 minutes, after which depending on the size of the seismic event, something like a gunshot can be heard. When trained, a person can distinguish between the various earthquake ratings.

When to Use Sonification Instead of Visualization

Some very general rules can apply when deciding what data in an application is sonified and what data is not. Typically sonification should be used when the data presented is simple, short, time based, or urgent. This is because the mind is able to interpret patterns of serialized sound events easier than patterns of serialized visual events and because sound is always heard, independently of the listener orientation with respect to the sound source [Deatherage72]. Conversely visualization is useful when the data presented is complex, long, spatial, or non-urgent; it allows the user to view everything in parallel, allowing the user to instantaneously change their focus to intake different aspects of the data as needed.

Requirements for Sonification Tools

Here we outline some requirements when designing a sonification tool. Included in these requirements are certain necessary components and features that are important to have in a sonification tool. Components include the following:

- Sound synthesis support for specification and control of the waveform.
- Control software to aid in manipulation of the produced sound. The controls should be accessible and should logically map the sound space in a way that makes sense.
- Analysis tools to extract significant elements from the data.
- Editing tools for basic manipulation of waveform or sequences of sound events.
- Signal processing utilities.

In addition to the components, there are features that the software should also exhibit:

- Portability. Tools should be consistent, reliable, and portable across many computing locations and platforms. This indicates that tools should use open standards as much as possible, limiting their dependency on proprietary features.
- Flexibility. Tools should be extendable enough to support new ideas when they are found. Controls should be appropriate to the data being sonified. Tools should be flexible enough to support many audio synthesis techniques. A simple interface should be provided to make the tool accessible to novices.
- Integrability. Sonification tools should work with existing visualization tools.

Methods of Sonification

There are many methods of sonification. The ones below are the most commonly used:

- **Auditory Icons** are single sound events that represent what they sound like in the real world. For example a grinding sound could happen when a user deletes a file from their computer, or a scraping sound could happen while dragging the file to a new location. An auditory icon is designed to be instantly familiar when heard, and correlates closely with the actual event that happened [AuditoryIcons].
- **Earcons** are tones, or sequences of tones that form the basis for building messages. In music, a short series of tones is called a *motive*. Earcons are constructed from a series of these motives. The advantage is that with many sound events, many parameters are available to manipulate such as pitch, tempo, timbre, and loudness. The motives can be combined or transformed to create more complex structures [Earcons].
- **Direct Simulation** is a method of mapping raw data directly to sound. Example uses of this method are described above in the seismology example. Direct simulation works well in cases where data values closely resemble audio phenomenon, and contain events that vary over time. For example, in Computational Fluid Dynamics Data (CFD) data, samples of pressure over time make a good candidate for playing directly to audio hardware. This technique can be thought of as placing a virtual microphone into the stream of data to sample the pressure at a given point [AuditoryDisplay]. Other parameters like flow direction, and velocity are difficult to represent with direct simulation, and require another method.
- **Dynamic Music** is a method in which music is used to represent data. Typically in dynamic music, music events are generated in response to data. Response can be to the data as it changes in real time, or as a preprocessed step. In dynamic music, it is desirable to force the music into something that is qualitatively listenable to most people. Typically this means incorporating elements that in music research have proven to provide listenability such as providing a sense of completion and phrasing.

As discussed in this chapter, the topic of sonification is important because it defines methods of utilizing sound as a medium to represent information. In this thesis we address several application domains and the ones that rely the most on sonification are virtual environments such as scientific or entertainment.

The following chapter describes the requirements for audio synthesis applications, including sonification, which lead into the design of Subsynth.

CHAPTER 3 REQUIREMENTS OF AN AUDIO SUBSYSTEM FOR INTERACTIVE APPLICATIONS

This document focuses on two genres of interactive audio applications: virtual environments, and music. Within each of these two genres, there are several areas that make use of audio with interactive capabilities: scientific sonification, interactive composition and performance, and virtual environment sonification. These areas will drive the requirements and design satisfied by Subsynth, and we will discuss each in turn in the following sections. Based on those areas, we will define the requirements of an audio subsystem for interactive applications.

Scientific Sonification

In the previous chapter, we discussed the reasons why scientific applications may need to *sonify* their data or present sound changes to the user to help understand the data. In the past, sonification has been done offline, but recent advances have allowed scientific sonification to be done in real-time [Bryden02]. This follows similar trends in scientific visualization. Scientific sonification may be done in response to real-time data or by allowing the user to probe areas of the data interactively.

Interactive Composition and Performance

Music authoring tools on computers allow the musician to input parameters in real time. This means that the user gets instant feedback about how the finished work will sound, thereby putting the human into the production loop. This is important because music is often a direct expression of its composer. Like any real musical instrument, a computer should respond in real time to provide this ability for expression.

Music composition and playback applications demand a high degree of flexibility, as do tone generators for performance and studio work. Not only do these applications need to have a high degree of polyphony (number of sounds able to play at a given time), they also need low latency

resulting in highly interactive timbres. Complex polyphony is important so that the user does not run out of simultaneously usable sounds too quickly. Low latency is important so that any user interaction performed feels natural and does not hinder artistic expression and creativity. Interactive timbres allow the composer to manipulate the tonal quality of the sound adding to expressiveness of an instrument.

Configurability is important so that the artist or sound engineer can change the system to fit their needs either statically as a setup phase or dynamically during performance or playback. An example of dynamic modification could be a filter sweep to change instrument timbre over time in a piece of music. An example of a static setup could be to choose an audio synthesis method, or to select the specific parameters for that method.

Virtual Environment Sonification

A virtual environment (VE) is one that gives the user a feeling of *presence*, the feeling of being there, by immersing them in some computer generated world. VEs are usually explorable and interactive geometric spaces. They are presented using realtime three-dimensional graphics and sound and can include force feedback, special input devices, and displays. Most modern games such as Quake [Id], Ico [SCEI], and Grand Theft Auto [Rockstar] provide examples of VEs. Fully immersive VEs are seen most often in scientific visualization centers and virtual reality research labs such as the Virtual Reality Applications Center (VRAC), Argonne National Labs (ANL), and National Center for Supercomputing Applications (NCSA). Examples of these immersive VEs are Cueva De Fuego, and the Nexus [VRAC].

In typical VEs, uses of audio involve several wave table sound sources linked to various events. For example, VEs may have audio file players for ambient music loop playback (wave or mp3 format for example), error tones, and other simple one-shot sound triggering. These uses are often less

intensive and have lower requirements than the highly dynamic use of sound in "Scientific Sonification" listed above. Collaborative VEs may also make use of streaming audio for distributed communication. For enhanced immersion, VEs may use filters on triggered sounds such as Doppler, reverb, and Head Related Transfer Function (HRTF) [HRTF] as a secondary means to present motion, room size, room material, and 3D position.

Requirements

Considering the application domains we have outlined, we now outline our requirements for an audio synthesis toolkit.

- **Configurable.** To support uses beyond an audio synthesis tool's original design, the system must be very configurable. For this, the system design should be partitioned into reusable, pluggable pieces where applications use only as many pieces as they need to get the desired results. In addition, the toolkit should reconfigurable while the application is running rather than limited to only one setup step at the beginning.
- **Useful Synthesis Methods.** The toolkit must provide a series of audio synthesis methods that make the toolkit instantly useable without the need for much user extension. At a minimum, we feel that it should offer wavetable and subtractive synthesis methods, both of which give a wide range of control to applications. Specifically, this means features such as one-shot sounds, looping sounds, and environmental effects such as reverb, delay, and filter.
- **Extensible.** To support future needs, the audio toolkit must be scalable and adaptable to new software extensions such as user defined audio synthesis techniques and must scale, adapt, and take advantage of underlying hardware such as CPUs or DSPs when available. Because future components may be added later, a common generic interface should be supported so that applications can take advantage of the new features.

- **Real-time Interactivity.** A primary feature needed by these groups is the ability to manipulate the sound's control parameters based on the user interaction. The sound system therefore must be flexible enough to offer many control parameters and be able to respond in a timely manner to the user's unpredictable actions. The real-time nature of the toolkit should be designed to operate as a *soft* real-time tool. VR and entertainment systems are often considered soft real-time [Manimaran01] [RealtimeVR]. This means that in order to deliver a highly interactive and immersive experience to the user, these systems must be highly interactive and engaging to deliver an immersive experience. As such they must have a high frame rate, good audio fidelity, and low-latency response to user input so to not make the user sick or confused [Pausch92] [RealtimeVR]. VR and entertainment systems are not (usually) considered *firm* or *hard* real time because occasional slowdowns or anomalies in these applications do not have a severely detrimental effect on users. For example, it is generally understood by these users that if they use their system to perform some other task, then the soft real-time task will probably be affected. The metric that these soft-real-time systems go by is *does the system present a convincing enough reality under normal conditions?*

- **Portable.** With potential for code development across several machine and operating system types the audio synthesis toolkit must be portable. Our applications range from tone generation for music and 3D games running on common PCs to scientific simulations running on powerful multiprocessor or clustered systems. We focus our solution to consumer level and professional workstation systems. This includes operating systems such as Windows, Mac OS X, IRIX, and LiNUX. Developing portable software for these platforms is easy if care is taken to write the software in terms of cross platform subsystems and programming language features. For our research group at the Virtual Reality Applications Center (VRAC), being able to support these platforms allows us to keep our VR software scalable between PC and high-end SGI systems, while looking to migrate to lower cost LiNUX clusters.
- **Consistency.** The sounds that the toolkit makes should sound the same wherever it is used to facilitate application consistency across computing locations.
- **Interface Accessibility**. The toolkit should have a clean application-programming interface (API) and be accessible from an application of the type listed above. This interface should be consistent with the real-time requirements, and cannot introduce latency or excessive computational overhead.
- **Runtime Location Agnostic.** The synthesis tool should be able to run within the same computer as the end-user application, or on a separate machine when extra performance is needed. This decision should be left to the application writer, so the tool should make no restrictions with regard to where or how it is run.

An additional requirement that we impose is the free distribution of the audio synthesis tool. This requirement is certainly optional, but is one that we chose to follow during the work of this thesis. We want audio developers and users around the world to be able to combine efforts to maintain and

advance it. To keep the integrity of the tool, it should go under an OSI approved license [OSI]. These licenses have undergone close scrutiny by the public, and in general allow standardized use, peer review, bug fixes, and feature submissions. With the requirements defined, we move on to investigate several tools already in use and discuss how they address these requirements.

CHAPTER 4 CURRENT TOOLS FOR INTERACTIVE AUDIO

Here we review existing audio tools that can be used for interactive real-time audio as described in the previous section. We have chosen to review a selection of those tools that are the most general, scalable and portable; they are also the existing tools that most closely meet the requirements presented in the previous chapter.

Musical Instrument Digital Interface (MIDI)

Musical Instrument Digital Interface (MIDI) defines a standard serial interface for controlling musical devices [Roads96]. MIDI does not transmit sound but instead transmits messages which can be used to control sound devices or devices completely unrelated to sound such as stage equipment. MIDI was designed in 1983.

MIDI is an asynchronous serial interface with a transmission rate of 31.25 kilobits per second (kbps). Each byte sent is 10 bits long, starting with a start bit, 8 data bits, and a stop bit. The MIDI transmission protocol is made up of messages. Each message is contained in an 8-bit string, although some message data can span these to become many bytes long. Typical messages that MIDI defines are note off, note on, aftertouch (key pressure), control change, patch change, pitch wheel, and system exclusive. System exclusive (sysex) is the message that makes MIDI somewhat extendable. Sysex defines a way for vendors to implement specific non-portable control methods for their devices. The problem is that this message type is not standard, and thus it has the potential to make applications control of tone generation inconsistent between MIDI devices.

Each MIDI message can be directed to a "channel". In MIDI, there are sixteen channels, and these are used as a sort of cache to store instrument state. Because setup of each instrument may take the device longer than the next note trigger, these channels are typically setup before any notes are sent. Each MIDI message then executes within the scope of that channel. There are some global

messages that execute outside the scope of any channel, but these do not make use of the channel (instrument) state [MIDISpec].

MIDI is an established standard that allows music control to be decoupled from the tone generator. There are a diverse range of tone generator devices available that support the MIDI protocols. Unfortunately, MIDI does not specify exactly what the tones sound like, which means that the resulting sounds produced by tone generators can vary dramatically. MIDI provides no method to configure specific audio synthesis algorithms; rather it depends on tone generators to define what algorithms are used. This hard-coded nature of current MIDI tone generators is a limitation. It means that the controlling application has only high-level control over the resulting sound. Any specific control over the sound through non-portable means such as MIDI sysex can be lost after migrating to another tone generator.

MIDI has some issues with addressing. Tone generators are only required to support sixteen addressable channels. The maximum number of addressable devices is fixed. Latency can be a limitation when accessing several tone generators at one time because devices are connected through daisy chaining, which means there will be some latency introduced as the MIDI message is retransmitted from device to device. These addressing and latency limitations can be overcome through use of a MIDI hub. A hub can fix the latency problems associated with daisy chaining because it can send the messages simultaneously from one hub rather than needing to retransmit over several hops in the daisy chain. A hub can also increase the number of devices available through multiplexing.

In summary, MIDI is good for most applications, but it fails our requirements criteria. The reason is that applications written for MIDI can either be complex or portable but not both. If they are portable, they cannot take advantage of every feature of the synthesizer because it would need to use sysex messages, which are device-dependent with contents not part of the MIDI standard and thus

are not guaranteed to be implemented by any two MIDI devices. Consider the case of sonification of many-dimensional data. If this application needs access to more parameters than standard MIDI has, then sysex will be needed to gain control of more synthesizer features. For these complex applications using MIDI portability and complexity is a tradeoff that we hope to avoid with our design of Subsynth.

CSound

CSound is a very flexible tool from MIT that allows user interaction through scripting and some input devices such as MIDI [CSound]. The tool supports both an offline renderer as well as a real-time renderer. CSound started in the 1960's with the Music 4 program written at Bell Telephone Laboratories by Max Mathews. This work coined the wave table concept and much of the terminology still in use by music researchers today [CsoundRef]. Followed by Music 4, was Music 4B, Music 360, and Music 11. With Music 11, control and signal processing was separated into distinct networks, the design of which still survives in the modern CSound.

CSound is written entirely in the C programming language [Kernighan88] and runs on any UNIX or Win32 machine. CSound defines a scripting language. This language can be used to control and specify music synthesis algorithms and actions. The two scripting languages available in CSound are for instrument definition and for score definition. The instrument scripts, having an `.osc` extension, allow a user to arbitrarily create instruments using mathematical notation (see Listing 8 in Appendix A). The score scripting language, having a .sco extension, allows control of the instruments with a series of time-based events.

CSound includes basic additive, subtractive, and non-linear synthesis methods. It also includes more recent additions such as phase vocoder, spectral data types, and granular synthesis. A MIDI converter is included allowing CSound to be run from MIDI files and external keyboards. Of interest

is the real-time support in CSound with run-time event generation (via the score scripts and MIDI files) allowing run-time sensing and response setups for interactive composition and performance.

CSound is currently only available as a script interpreter having no accessible application programming interface (API) useful to most virtual environment applications being developed today. The software is written in C, but the API is not exposed or supported for use by C programmers. Lastly, CSound has a non-OSI license that only allows educational use. Any derivative works or tools that use CSound would have to have the same license. This limits CSound from widespread use particularly in the commercial sector.

Supercollider

Supercollider is an environment for real-time audio synthesis [SuperCollider]. It is a software-based audio synthesis toolkit that only depends upon the computer system's main processor and the PCM audio port on the sound card. SuperCollider provides a new programming language that derives its structure and syntax from both Smalltalk and C. The programming language features garbage collection, functions, and a small system for object oriented classes. Other utilities provided in SuperCollider are a GUI builder for creating patch control panels, a graphical wavetable and breakpoint editor, MIDI control, and a very large collection of signal processing and audio synthesis functions.

SuperCollider provides several classes of unit generator available as objects in the scripting language:

- **Unary Operators:** for every mathematical function (such as sine, cosine, floor, absolute value, negate, square root, etc.)
- **Binary Operators:** such as +, -, *, /, min, max. round, etc.
- **Oscillators:** such oscillators as wavetable, sine, impulse, saw, pulse, formant, etc.
- **Noise:** such as white, pink, brown, clipped, etc.
- **Filters:** such as resonator, one pole, two pole, low pass, high pass, band pass, band reject, butterworth, integrator.
- **Controllers:** such as envelope gen, trigger, trigger delay, gate, sequencer, etc.
- **Amplitude Operators:** such as compander, normalizer, limiter, stereo and quad pan
- **Delays:** such as 1, 2, and N sample delay line with or without interpolation, comb filter, all pass, multi tap, time domain pitch shift, ping pong.
- **Frequency Domain:** such as FFT, inverse FFT.
- **I/O** Adaptors: read and write adapters on top of signal buffers, disk, and hardware audio ports.

From the list above, we can see that the objects available by SuperCollider are varied in purpose. With this array of methods available, SuperCollider is well suited to anyone using additive, subtractive, AM, FM, or granular techniques.

SuperCollider's strengths are its high level language (See Listing 9 in Appendix A for an example), which makes it very useful for quick prototyping and ease of use by people not used to lower level programming languages such as C or C++. Also the huge number of signal processing units included makes it very flexible for creation of many audio synthesis methods. SuperCollider included tools for graphical user interface creation, which makes it easy for users to interact with the algorithms they create. SuperCollider only runs on a Power Macintosh OS 9, with no plans to port to Windows, or any UNIX workstation. There is no provision to allow extension or addition of unit generators. SuperCollider only allows programming in their scripting language and provides no bindings for C or C++ libraries or applications. This makes it very difficult to integrate into the types of applications listed in the requirements section. At the beginning of this thesis work, SuperCollider

was a commercial product. Now it is free, but the source code is not available yet, and the license has not yet been announced. At this point, there has been no mention of an OSI-approved license.

Virtual Audio Server (VAS)

The Virtual Audio Server (VAS) is a toolkit created at the Naval Research Laboratories (NRL). VAS was designed to facilitate exploration of the problems associated with designing virtual sonic environments (VSE) [Fouad00]. The focus in this toolkit is on the administrative side of setting up system hardware such as speaker arrays and driving them with the appropriate audio signals from software. As such, VAS includes pluggable "localizers".

VAS defines a localizer as some transform than takes an audio signal as input, processes it, and outputs the resulting signal to the speaker array. Ideally the transform is such that the audio output sounds to the human ear as if each audio source it emits from a point in 3D space. VAS includes 3 localizers by default:

1. **Panning**, which is simple and inaccurate. The panning technique calculates the sound level at each speaker in the array and attenuates the sound source's audio signal by that amount for each speaker.
2. **VBAP**, or Vector Base Auditory Panning, is more accurate and works in three dimensions. VBAP is a three-dimensional spatialization technique that uses speaker panning. The technique chooses three speakers out of an arbitrary number of speakers and pans an audio source between them. This technique results in spatialization of an audio source in three dimensions provided that the speaker array is three-dimensional. Because this technique simulates the sound field and offers no audio synthesis capabilities, effects such as propagation of sound through air cannot be recreated. VBAP works best with a large number in the speaker array [VBAP].

3. **HRTF**, or Head Related Transform, is the most accurate but also most expensive. HRTF uses a technique called Perceptual Synthesis to render sound that closely models what a user's ears would hear [HRTF].

Each localizer is selectable by the user; also the user is able to supply their own localizer if they wish to experiment.

VAS was implemented in C++ and runs on the IRIX operating system by SGI. VAS provides two libraries, one for local execution, and one for execution using a sound server. VAS has many good features, in particular for VSEs. It allows researchers to try different algorithms and loudspeaker configurations with no change to the VAS framework. VAS is scalable from headphones to small speaker arrays to arbitrarily large speaker arrays. With this built in functionality, VAS is very flexible. VAS is limited by IRIX's AL library, which only supports up to 16 sounds. VAS is primarily a research tool, which could imply that it is not ready for production use. Also at the time of this writing no version was available for download, and the license is unknown.

Virtual Sound Server (VSS)

The Virtual Sound Server (VSS) is a proprietary audio toolkit developed at the National Center for Supercomputing Applications (NCSA). According to [VSS], "VSS is a platform-independent software package for data-driven sound production controlled from interactive applications." The work in VSS was derived from a tool called HTM, which is a framework for real-time sound synthesis controlled via network. As a result, VSS is also controlled by network connection. VSS offers control of software synthesis, simple sample playback, and provides abstraction to other synthesizers such as MIDI, Max, and Open Sound Control (OSC). VSS claims to be platform independent but currently only offers SGI IRIX and Linux versions.

To use VSS, a server process must already be running. A client application connects to it over the network, after which the client can direct the server to perform sound related tasks. For each sound the client allocates, the server gives back a handle to the sound, which the client uses to manipulate that sound. This practice of referring to sounds by handle is common in most sound systems whether they are accessed over a network or not. VSS offers a higher-level construct on top of each sound or group of sounds called "actor". An actor is also manipulated using a handle, and the actor is used to specify particular audio synthesis methods. In effect, actors are used to group and position sounds. Like sounds, actors can accept messages that define control parameters. The VSS client or other elements such as actors can receive these messages. A data flow is defined so that actors can send messages to other actors without client intervention. The client ahead of time can define these actions. Actors process real-time input and output in the form of audio or MIDI. All network connections in VSS are masked by higher-level client-side function calls.

VSS defines a data format to facilitate data driven applications called the AUD format. This allows separation of sound logic from the main application code. Benefits of this are modularity and the ability to change the sound logic without recompiling the application. The AUD format maps application events to sound procedures.

In conclusion, the VSS is a high level toolkit aimed at virtual environment application programming. VSS offers a range of audio synthesis methods under a fairly high level interface controlled by event messages. It is limited by the number of platforms it supports, currently running on only two types of UNIX operating system. For true portability, application designers will need to modify VSS or look elsewhere. Another limitation is that VSS only offers client/server control via network. This limits VSS to high-end applications needing multiple computers, or one computer with two processes communicating via the operating system's TCP/IP stack. This is less efficient than direct access of the sound in the same application process space. The VSS programming interface is

general enough that the VSS client implementation could theoretically be extended to include the server communicating and running in the application's process space. However this option was not provided at the time of this writing. VSS does not offer an interface for defining arbitrary audio synthesis algorithms, so it is not extensible by developers. Some applications may need a finer grained toolkit offering more direct control with lower level access. The high level aspect, while a benefit to some, could also be viewed as a limitation that restricts applications into limited methods of sound manipulation. VSS is specialized for event based message triggering of pre-programmed audio synthesis methods. Applications needing to redefine synthesis methods or do things outside of the scope of VSS may have difficulty. Even though VSS has limitations, it is still very useful for typical high-end virtual environments and sonification applications developed for the Linux and IRIX operating systems. Lastly, the largest limitation with VSS is the license, which currently is restricted to single user non-commercial use, cannot be redistributed, and derivative works cannot be publicly shown.

Digital Instrument for Additive Sound Synthesis (DIASS)

The Digital Instrument for Additive Sound Synthesis (DIASS) is a tool created jointly by Argonne National Laboratory (ANL) and University of Illinois at Urbana/Champaign (UIUC) [Kaper98]. DIASS is an audio synthesis tool used to sonify scientific data on Argonne's high performance supercomputer at the Center for Computational Science and Technology (CCST). It operates with additive synthesis, a technique that can create arbitrarily complex sounds, and it can be parallelized across multiple processors.

Using additive synthesis, DIASS is able to offer a very flexible means of sonification. As we mentioned before, additive synthesis can overwhelm the user with the number of data inputs needed.

This also means that additive synthesis can sonify very complex processes containing many degrees of data.

DIASS consists of two parts: an editor and an instrument. The editor provides a GUI for preview and authoring of individual sounds and a script reader, which takes data input from a script. The instrument responds to messages from the editor. The instrument is capable of playing an arbitrary number of sounds where each sound can be made of an arbitrary number of partials. Each partial can be controlled up to twenty-five ways. Some of the controls are static such as start time, duration, and phase. Other parameters are reverb and echo. Some of the controls are dynamic and include envelope, panning, amplitude modulation (tremolo), and frequency modulation (vibrato).

Because additive synthesis can present the user with a huge number of inputs to control, DIASS offers macros to perform global operations over collections of sine waves. For example, adjusting the loudness of a sound should affect all partials. Also because DIASS primarily uses additive synthesis with many voices, it is computationally intensive and requires a large amount of memory. The idea behind the design of DIASS is to use as much computing power as is available. The instrument comes in sequential and parallel versions. The parallel version uses MPI, a message-passing library. MPI facilitates parallel computing across many *nodes*, which are computing units such as computers or running processes [MPI]. Parallelism is implemented at the sound level, not at the sound partial level. A bottleneck exists with this approach in that after every sound is computed, each node must deliver the result to a single "mixer", a computer that combines each sound together for final output for listening. DIASS does not currently run in real-time, but this is a feature that will be added in the next release.

Strengths of DIASS include a massively parallel sound renderer utilizing additive synthesis, which is an audio synthesis method known for its ability to handle many data inputs simultaneously. DIASS is implemented in C, and this could be a potential limitation since audio synthesis algorithms

are object oriented by nature. A language like C++ could provide built in language features that would better represent the design of DIASS. The next version of DIASS promises to be implemented in C++. The main limitation of DIASS is that it does not currently run in real-time. The parallel version of DIASS would not be suitable for most users—especially individuals—because of the size and/or number of machines required to run it. DIASS only supports additive synthesis methods, plus a few extra techniques such as wave modulation to manipulate the additive synthesis generated sounds. Wavetable and other forms of audio synthesis are not advertised features. Additive synthesis is very costly in terms of processor usage compared to other synthesis techniques such as wavetable and modulation synthesis. The reason is that additive synthesis methods require as many simultaneous voices as the degree of complexity of the waveform. In short, additive synthesis does not scale well on a given system when complex sounds are needed. Most VE and music applications need complex sound. Lastly, DIASS is not available for use outside its authors and their affiliates. In conclusion, DIASS is a very flexible tool for additive synthesis, however it does not offer other forms of audio synthesis making the tool not useful in certain application domains.

OpenAL

OpenAL, the Open Audio Library, is an effort to create a vendor-neutral, cross-platform API for interactive three-dimensional (3D) spatialized audio [OpenAL]. Two-dimensional concepts, such as panning, are considered "legacy" and are not supported. OpenAL's primary application audience is 3D entertainment VEs (such as games) and other popular multimedia applications. OpenAL is currently supported by Creative Labs [Creative], but originally began with Loki Entertainment, a Linux-based game company.

OpenAL aims to be the OpenGL of audio. OpenGL is a standard real-time graphics API that abstracts every detail of computer graphics functionality needed by 3D applications. OpenGL

provides a vendor-independent open standard that anyone may implement. With its well-designed API and open nature, OpenGL is an industry standard and is used in thousands of applications ranging from commercial to research. Similarly, OpenAL attempts to emulate OpenGL by creating an open standard audio API that anyone may implement. There is a UNIX version (Linux and IRIX) created by Loki Entertainment, and a Win32 and Mac OS X version created by Creative Labs. The version by Creative Labs includes support for their EAX environmental audio function [CreativeDev].

A typical application will begin to use OpenAL by opening an OpenAL sound device. Then a context is allocated and associated with the device. With the context allocated, the programmer can create sound buffers and sound sources. The sound sources are the "sound objects", and the buffers are the source data to the objects. The sound objects in OpenAL support point and directional 3D rendering that have the ability to play once, or loop continuously. Object state can also be manipulated to affect sound object parameters such as distance attenuation, orientation, volume, pitch, and filter.

OpenAL provides an object-oriented C programming interface for spatialized sound objects. The interface allows a programmer to specify sound object positions and a listener in three dimensions. Underneath the API, OpenAL is a multichannel processing system for the synthesis of a digital audio stream. It offers a fixed pipeline of digital signal processing algorithms to send wavetable data through. The digital oscillator in OpenAL generates an audio signal from wavetable data and has a few options that the user can set. For example, the user can set the speed of the wavetable traversal to affect the pitch of the sound source. The user can also confine the traversal of the wavetable to once or infinite, for a "one-shot" or a looping effect, suitable for sound effects or ambient music and sounds respectively. After the wavetable sound is generated, it travels through a series of processors

for 3D spatialization including environmental effects, reverb and echo, Doppler, and distance attenuation.

A strength of OpenAL is its multi-platform support. It covers almost every major platform including, Windows, Macintosh, Linux, and IRIX. Another strength is its open sourced, usable sample implementations. OpenAL's license has been designed so that anyone may implement a version of it. As an added benefit, Loki and Creative labs offer free LGPL [OSI] (OSI compliant license) sample implementations. OpenAL is very useful to simple virtual environments (VE) such as most VE entertainment applications. It provides wavetable synthesis with a small set of processing algorithms useful to 3D spatial audio rendering. Limitations of OpenAL are that it only supports wavetable synthesis and does not support traditional audio rendering; its philosophy is hard-coded to 3D spatialized wavetable audio. Considering OpenGL, OpenAL does not compare as a generic audio tool. It is specialized to the, albeit very common, use in simple spatialized sound object triggering. OpenAL cannot be extended to support more advanced synthesis methods, nor can it support restructuring of the audio pipeline. In conclusion, OpenAL has its place as a very useful tool, yet is restricted to a subset of application domains listed in the requirements section of this paper. For sonification, OpenAL is limited by its support of only wavetable synthesis. For interactive composition and performance, OpenAL does not offer enough features in their processing pipeline to handle the range of dynamic effects desired in music.

PortAudio/PABLIO

PortAudio [PortAudio] is a very simple developer tool for streaming digital signals to the audio port of the computer. It offers a cross platform C API for audio streaming. The features of PortAudio are that it is fast, has a very small code footprint, and runs on every major computing platform. Under Windows, it supports ASIO (very low latency) [ASIO], DirectSound (low latency), and

Windows Multimedia Extensions (high latency). Under Unix it supports the OSS API that is also implemented by several system audio drivers including ALSA. Also supported are Apple's Mac OS X Core Audio and Sound Mgr for OS7-9, SGI's IRIX AL library, and BeOS.

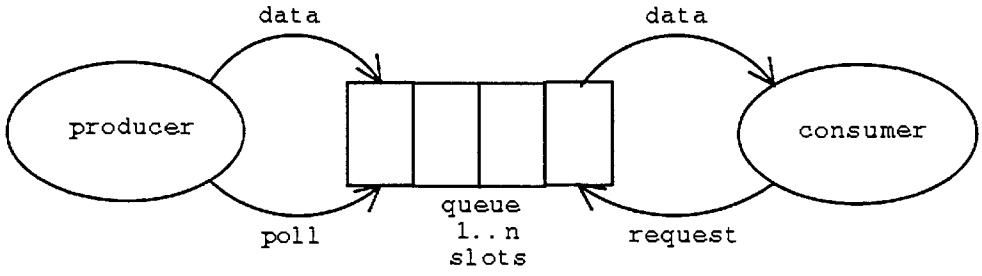

Figure 18. PortAudio Producer-Consumer diagram.

PortAudio operates with a callback architecture. With this method, the user registers a function with PortAudio, and whenever PortAudio needs more data, it calls the function to get the data. Data requests happen asynchronously to the execution of the application code, in a producer-consumer fashion (Figure 18). In producer-consumer, PortAudio is the consumer, and the application is the producer. PortAudio asks the application for more data asynchronously as needed. The application must either synchronize with this asynchronous callback process, or it needs to be written such that all audio code exists in the callback function. PortAudio also provides a blocked polling interface, called PABLIO (for **P**ort **A**udio **BL**ocking **IO**), which the user calls repeatedly similar to the network sockets `send()` function. Currently this only works in blocking mode, making it unsuitable for use in a single-threaded graphics application. The time waiting on the blocking call could seriously impact the draw and calculation update rates needed for the application. One solution could be to reduce the audio block size sent per frame. This is not a good solution since it could induce dropouts (data starvation at the audio hardware) if lowered too far or if the system needed to do some unexpected work such as during resource loading. In short, the first callback style interface is usually the best and is PortAudio's default interface. There is a third option that extends the idea of the PABLIO method and that is to make an interface to PortAudio that is non-blocking. A non-blocking

interface to PortAudio has been developed by the author and will be presented in the Subsynth section of this paper.

PortAudio can be especially useful for real-time software audio synthesis applications. Because of its low latency and overhead, it is a good low-level, cross-platform interface to audio hardware. PortAudio was developed by people on the music-dsp mailing list, a group dedicated to music digital signal processing. This tool is currently in use by many software synthesizers used in music composition tools such as Twelvetone Systems' Cakewalk [Cake], and Steinberg's Cubase [ASIO]. The software synthesizers are typically implemented as VST (Steinberg) or DirectX (Microsoft) plugins developed by third-party developers. Plugins are components that offer a common interface useful among a variety of software that support them. Also several independent open source composition tools and sound utilities use PortAudio.

Strengths of PortAudio include cross platform support and low latency support on certain platforms. PortAudio would make a very good audio port hardware abstraction for higher-level audio toolkits. However, PortAudio has some limitations. The callback interface might be awkward for certain uses, but fortunately PABLIO is included for polling style IO. Since PABLIO only supports blocking reads and writes, an application needing to do many other things may suffer in performance. The final limitation in PortAudio is that it only handles raw PCM audio streams. No higher-level audio rendering primitives are available. PortAudio will not work as-is for audio synthesis, but it will work well as a port abstraction for audio synthesis tools such as Subsynth.

Several tools have been investigated for their use in sonification of VEs, scientific data, and musical applications. PortAudio does not solve the job, but it looks like a good tool to use to abstract the audio for cross platform support. The other tools, while they have their strengths, are all specialized to their own goals and have one or more features that cause them to be unsuitable for our needs as defined in the requirements section (see chapter titled "REQUIREMENTS OF AN AUDIO

SUBSYSTEM FOR INTERACTIVE APPLICATIONS"). The next section introduces Subsynth, a new tool that attempts to address these issues.

CHAPTER 5 SUBSYNTH

Subsynth is a subsystem for sound synthesis designed for use by many types of higher-level applications such as virtual environments, interactive music composition and performance, and scientific sonification. It balances generality, performance, scalability, and portability with the goal to provide consistency across computing locations, general use across many application domains, and application survivability with the ability to adapt transparently to new hardware. The design of Subsynth is inspired by the unit generator concept (Figure 19) described earlier in the section "Unit Generator Language". We believe that a synthesizer designed with the unit generator concept will make an interesting research tool as well as provide a useful generic layer upon which higher-level audio applications and libraries could be built. We have chosen to build Subsynth in software in order to decouple the hardware from the toolkit. This provides the ability to move transparently to new hardware and operating systems. Here we will present the design and implementation of Subsynth, which currently offers the core audio synthesis framework definition as well as concrete extensions able to be used for audio synthesis today.

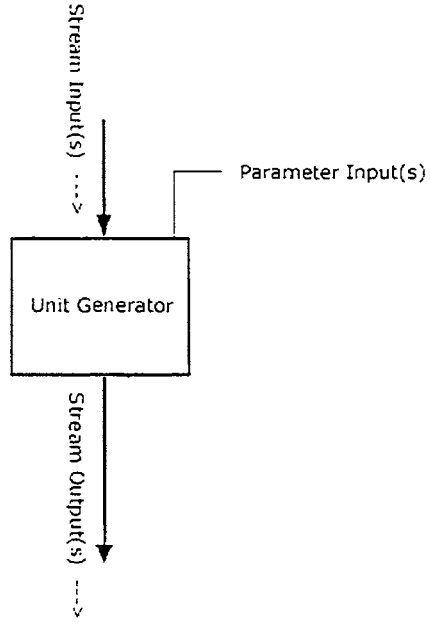

Figure 19. A Subsynth module.

Conceptual Design Approach

Creating software architectures that support flexible configuration can be difficult. Inventing something completely new is not practical nor will it always produce the best design. We can look at the unit generator concept described earlier (Figure 19) for inspiration. Modeled after unit generators, each Subsynth module has n input terminals, and p output terminals. Terminals pass data in a common format for compatibility between modules. Successful execution of the unit generator concept can be seen in existing hardware modular analog audio synthesizers such as the one shown in Figure 20 and in existing user-level music composition tools such as [Fruityloops] and [Propellerheads].

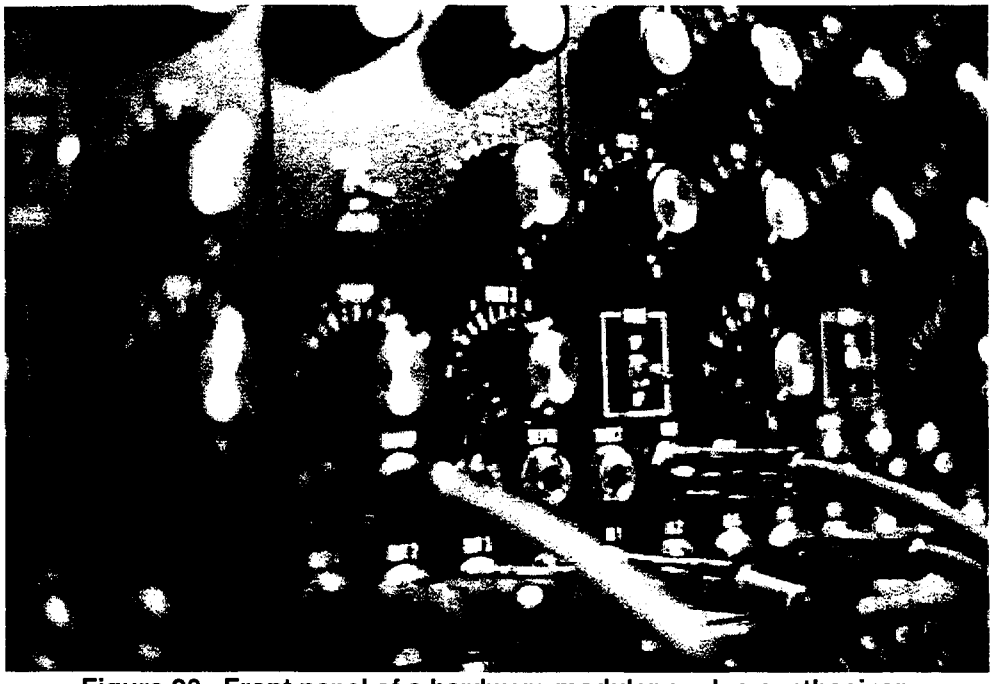

Figure 20. Front panel of a hardware modular analog synthesizer.

The original hardware modular analog synthesizers were designed in small reusable units (Figure 21) hooked together with patch cables (Figure 20). In these figures, the patch cables connect the input and output terminals, while each module has static parameters that can be adjusted manually with knobs. The signals traveling between these modules are in a standardized format so that

interfacing can be done generically without regard to the type of signal. This powerful design even allows audio outputs to be used as control inputs. This means that a signal can be modulated by any other signal.

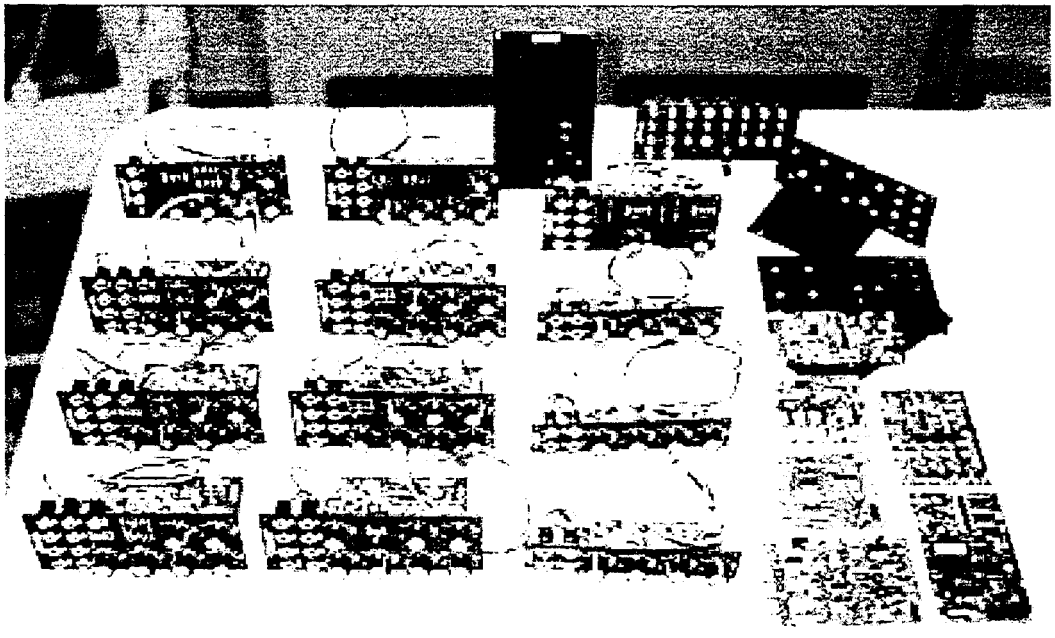

Figure 21. Modular analog synthesizer unit generators.

To support the interoperability between the various unit generators, input and output terminals of each unit are standardized. This unit generator architecture is very flexible and open, designed for arbitrary user configurations, generic interoperability, and creative freedom for extension and design of new audio synthesis methods. Building on this powerful concept, we chose to design Subsynth in a similar vein.

The Subsynth system design consists of three components: core audio framework implementing the unit generator concept, configuration allowing methods to specify audio synthesis networks, and task management used for execution of the configured audio synthesis networks. In the next sections, we explore the design and implementation of these three components.

Core Audio Framework

This part of the design seeks to solve most of the functional goals set out in the requirements section. It covers unit connectivity and data transport and is what provides the generic configurability we desire.

The audio framework in Subsynth is based on the unit generator concept. It allows arbitrary audio synthesis algorithms to be built, allowing Subsynth to serve as a very customizable audio tool. The unit generators in Subsynth are called *modules*. Just like theoretical unit generators, Subsynth modules take continuous and static input and give continuous output. Continuous input can be thought of as the audio or control signals, while static input can be thought of as parameter setup. Some modules are for generation of signals. Others are for transformation of a signal through methods as simple as inversion or summation and as complex as filtration or convolution.

Each module can have any number of input and output terminals, or connecting points, as well as any number of static parameter settings (Figure 19). Terminals may be connected together using a `Connection` (Figure 22), which acts as a sort of glue to connect two terminals. Terminals are simply connecting points, while the Connection is the actual medium where data travels. The connection of terminals allows arbitrary setup of a *sound network*, a graph resulting from all connected modules. After connection, data is propagated to and from the terminals using a storage queue located in the `Connection` object that may be filled or emptied by either `Terminal`. This signal transport medium serves both audio and control data purposes. Module parameters are then accessed and set through a generic interface so that users of Subsynth do not need to rely heavily upon interfaces that are proprietary to each separate module.

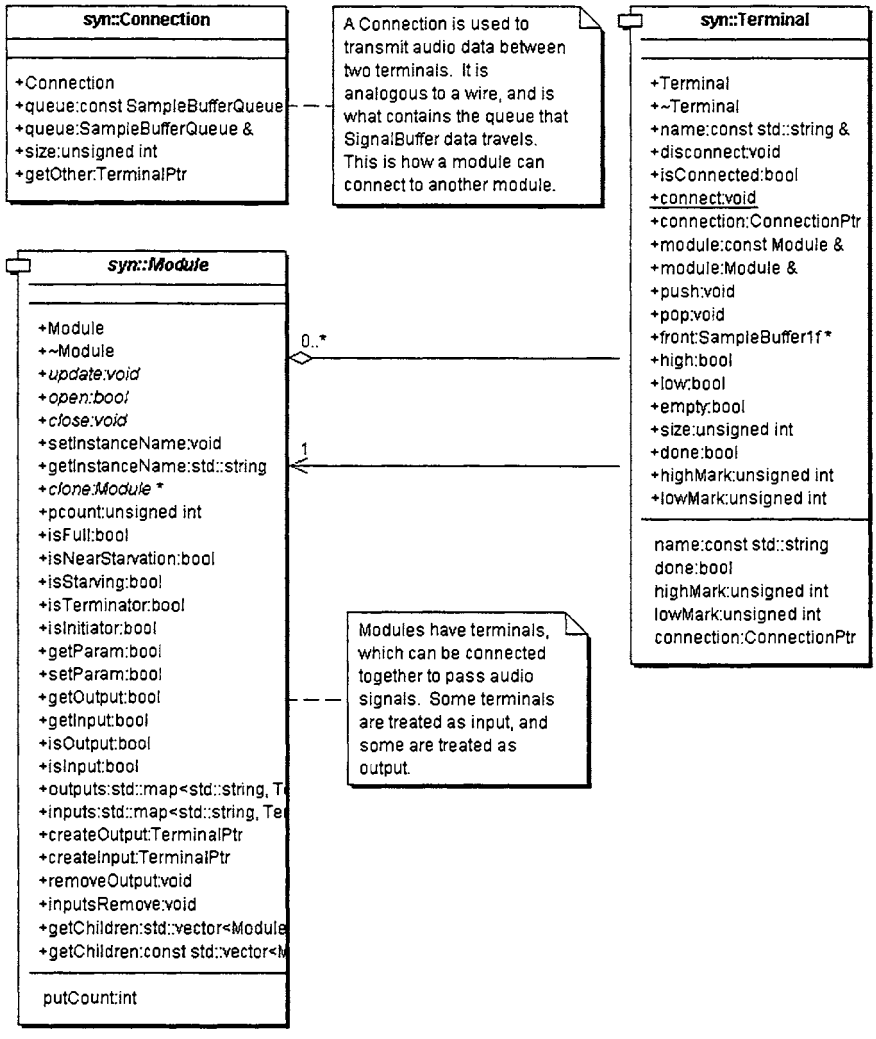

Figure 22. Subsynth unit generator design [UML].

Conceptually, `Module` is a black box containing a signal generator (source), processor, or an output (sink). Signal generation could happen by a mathematical function or a procedure (for example: sine, cosine, saw, noise, ADSR envelope), from audio samples (wave, mp3 file), or even from a network stream. Processing units could be defined in many ways. Some common filters include DC offset, gain, pitch shift, reverb, Doppler shift, localizers for 3D positional sound, and distortion [Steiglitz96]. An output sink represents a route to the computer's sound hardware or possibly to an output file. With

the basic audio generalizations in Figure 19, and Figure 22, Subsynth's users will be able to specialize their own `Module` types as needed.

Configuration

To specify and arrange sound networks, we did not want to tie the user to one method. Therefore, configuration in Subsynth is decoupled from the audio design. The audio design by itself already provides a well-defined API for configuration allowing direct connection of modules using native C++ method calls. This gives the programmer fine-grained access to details when needed. In addition we want to facilitate higher-level configuration, such as through scripts, which allow a more data driven method to specify arrangements of modules. We call these higher-level components for configuration *builders*.

Builders are objects that know how to construct an audio synthesis network. A builder should be thought of as a factory [Gamma95], which is basically an object that knows how to create something based on some input request. Builders have a one-way dependency upon the audio design so that new builder designs are possible without affecting the existing toolkit. This decoupling is useful to users who may want to implement their own builder objects to extend or replace functionality with their own.

Builders are also useful so that audio synthesis networks may be saved to and loaded from disk. This has important uses: authoring tools such as the one shown in Figure 23 could be built to allow users to create new audio synthesis algorithms and save them for use later. Audio applications can then load these definitions instead of "hard code" the algorithm in software. This provides flexibility and helps to keep a separation between the application code and the data that defines a particular

audio synthesis algorithm. This idea facilitates "data-driven" design [GameGems1], which is important when creating complex applications.

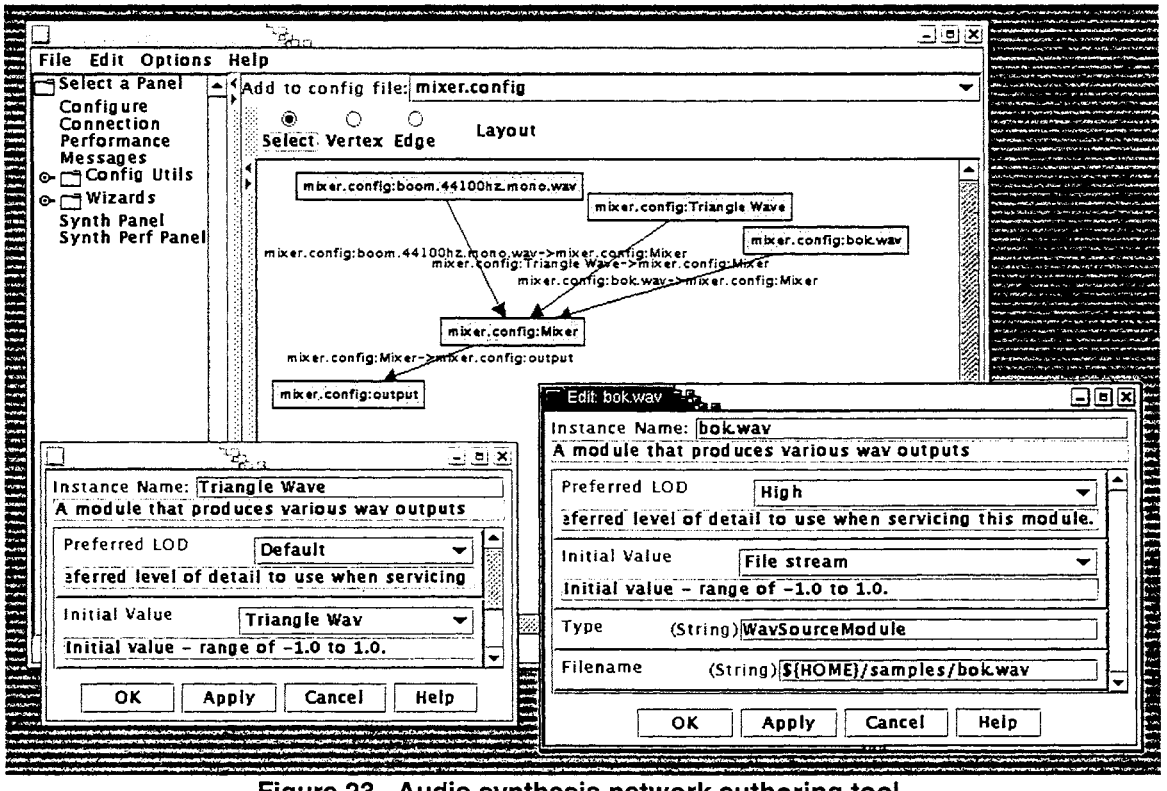

Figure 23. Audio synthesis network authoring tool.

Task Management System

We have designed a concept called the *runner*, which is responsible for executing the modules in a given synthesis network. The runner is decoupled from the main audio system in case the user would like to use or implement a different solution. For example, new runner systems can be customized to operate the synthesizer optimally on inexpensive consumer PCs or high-end workstations and supercomputers. The task management system also includes utilities for performance measurement that could be used by the scheduling system and could be displayed to evaluate a particular scheduling algorithm.

Implementation

In this section, we present the specific implementation of the design discussed above. First, we present external subsystems we decided to rely upon in order to abstract Subsynth away from the system hardware and the operating system. Next we present how we implemented the unit generator design, which is our core audio framework. Finally, we present our own specific extensions to the core that we include with the toolkit that ultimately provide the features that users will find useful about the toolkit.

Technology Choices

Part of our initial research work for the implementation included a search of available technologies to use under our system's design as outlined above. We knew early on that Subsynth should be as portable as possible. Therefore it was understood that Subsynth would need to sit upon, or *depend* on, some system abstractions. From the requirements, we needed the following:

- Threads
- Audio Output
- Configuration File Reader

As we began the design process we evaluated many system abstractions. We selected the following specific implementations:

VaPoR

VaPoR is a portable runtime layer that comes with the VR Juggler project [VRJuggler]. It abstracts system threads, NSPR [NSPR] threads, BSD sockets, and native serial port IO. We choose VaPoR because of our previous experience with it and other sync and threading abstractions. VaPoR offers a C++ interface on top of several subsystems, which provides an easy to use interface with the most portability.

PortAudio

PortAudio attempts to abstract an audio system and currently provides implementations for each of the major consumer platforms: Win32, MacOS, IRIX, LiNUX, and other UNIX platforms. PortAudio has a simple streaming audio interface and is in use by many real-time audio tools. We chose PortAudio because it is very small and efficient compared to other streaming interfaces we looked at (OpenAL). In our tests, PortAudio had noticeably less latency and less noise than OpenAL.

CppDOM

CppDOM [CppDOM] is a fast, lightweight XML [XML] tokenizer. It offers an intuitive C++ DOM (Document Object Model) representation and is very easy to integrate with a project. Subsynth uses XML files to parameterize attributes. Parameterization of attributes away from the code allows them to be changed without recompiling. We chose CppDOM because it is very lightweight and offers an easily traversable DOM tree in memory. This allows us to store configuration information in the DOM tree structure if needed. Other options we looked at were the VR Juggler JCCL [VRJuggler], Xerces C [Apache], and writing a custom configuration file parser. Xerces and JCCL are too complex for our needs, and writing a custom file parser had no benefit over CppDOM.

These three tools are the only external subsystems that Subsynth depend upon (besides native C++) and are referred to as Subsynth's *dependencies*. Subsynth is implemented only in terms of these dependencies and C++. The use of C++ and these tools facilitates the portability of Subsynth.

Core Audio Framework

Implementing the Module, Terminal, Connection paradigm was a matter of filling in the code behind the interface designs presented in Figure 22. For example, each module has an `update()` method that is invoked to cause the module to perform a block of processing on its terminals. See Listing 1 for an example of what a typical Module `update()` method looks like. Update is a

method that belongs to every module, that when called attempts to do a block of processing for that module's terminals. Modules also have open and close methods, as well as methods for accessing and configuring terminals.

```
void SinkModule::update()
{
   // if connected, then suck data from the queue
   // and throw it away...
   if (mMonoAudioInput->isConnected())
   {
      unsigned int size = 0;

      if (!mMonoAudioInput->empty())
      {
         SampleBuffer1f* read_buf = mMonoAudioInput->front();
         size = read_buf->size();
         mMonoAudioInput->pop();
         SampleBufferRepos::instance()->putback( read_buf );
      }

      // record how much was processed...
      this->setPutCount( size );
   }
   else
   {
      this->setPutCount( 0 );
   }
}
```

Listing 1. A simplified update() procedure.

In implementing the `Module`, `Terminal`, and `Connection` code, several internal issues appeared: how to represent the digital signals, how to efficiently allocate blocks of memory for signal data, how to allow asynchronous processing of modules in order to support multiple processors. Next we present how these implementation issues were addressed. The Producer/Consumer pattern is how we addressed asynchronous processing, the FlyWeight pattern is how we addressed fast allocation of large objects, and SampleBuffer is how we represent our data signals.

Producer/Consumer Pattern

One problem we had to solve was how to transport the data between modules. Since we want to be able to execute modules in any order and in parallel with other modules, we know that the module will need some way to cache the data for its connecting module to receive asynchronously. In distributed computing, a common design pattern to solve this problem is the Producer/Consumer

pattern [ProducerConsumer] (Figure 24). Design patterns describe simple and elegant solutions to specific problems in object oriented software design [Gamma95]. In Subsynth, we have one producer module and one consumer module, and we use a queue as a drop/pickup point for the audio signal data transferred between the two. To give us control of production we use a *watermarking* system. The producer module polls the queue's watermark, if not high then the producer will push data into the queue. The consumer can asynchronously poll the queue for data, if the watermark is not too low then it will obtain the object for reading.

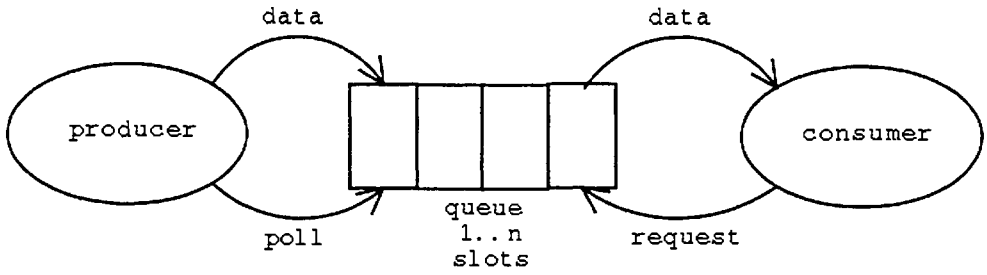

Figure 24. The producer consumer pattern.

The Producer/Consumer pattern coordinates the asynchronous addition and removal of information or objects. Generally, the Producer/Consumer pattern uses the Guarded Suspension pattern [GuardedSuspension] or the Balking pattern [Balking] to manage the situation of a Consumer object wanting to get an object from an empty queue. In Guarded Suspension a method call suspends execution until a precondition is satisfied. In Balking a method simply returns if a precondition is not satisfied. The implementation of the Producer/Consumer pattern in Subsynth is called `SampleBufferQueue` (Figure 25), and it uses the Balking pattern in order to give up processor time for other tasks to execute. The precondition we use in our Balking implementation is a watermark system so that queues do not become too full or too empty. The object in Subsynth that is produced and consumed from the `SampleBufferQueue` is called the `SampleBuffer`, which we explain in the next section.

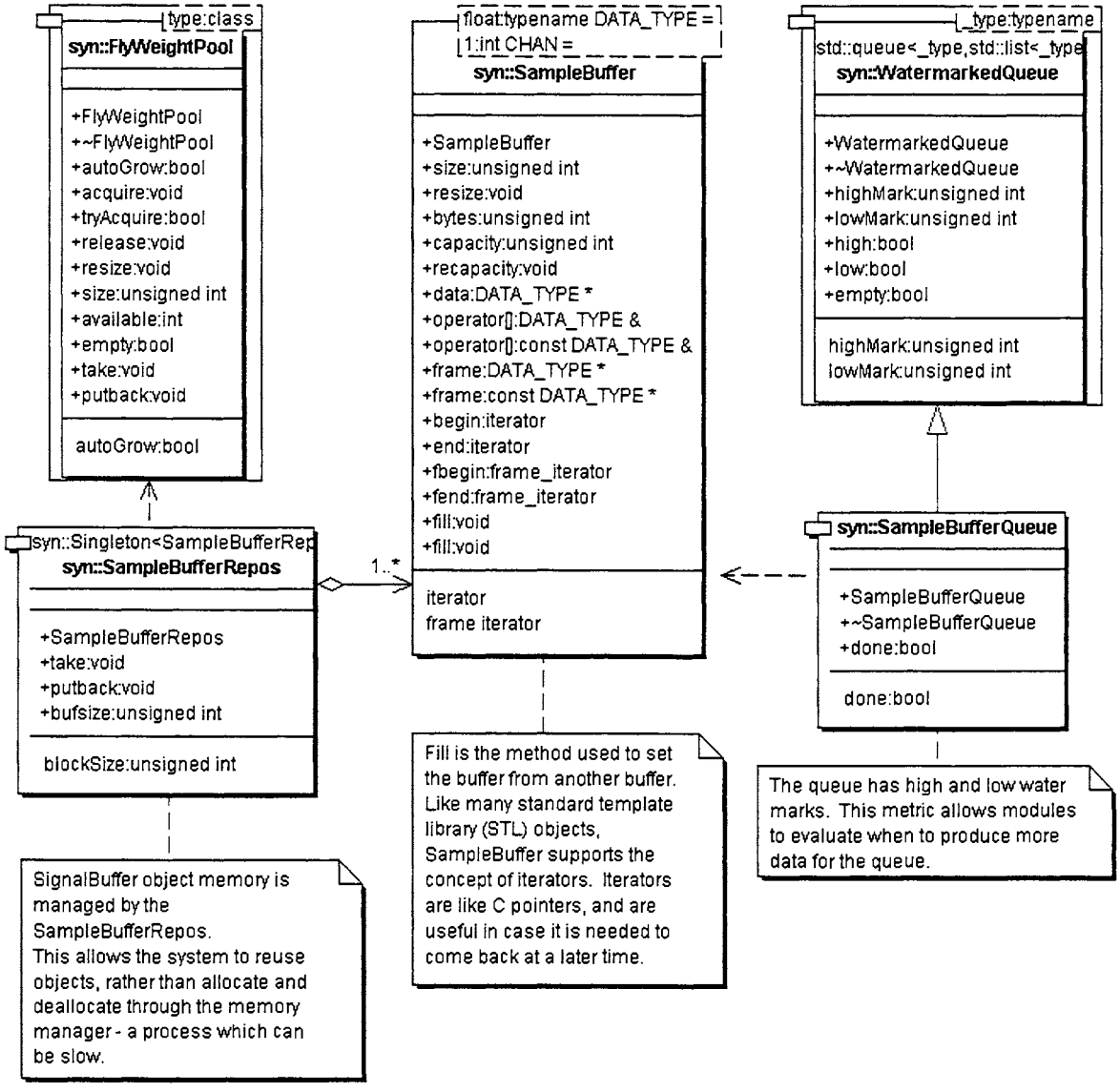

Figure 25. Producer/consumer implementation in Subsynth [UML].

SampleBuffer

The `SampleBuffer` in Figure 25 is the basic medium in which we store digital audio and control signals. It represents a *block* of audio signal samples. When a module looks for input to process, or when it generates some output, it produces (or consumes) a `SampleBuffer`. Using a block of data rather than individual sample values introduces some latency, but it increases processing efficiency because the overhead of function calls, conditional expressions, and synchronization points

become amortized over the length of time it takes to process the block. The longer the block, the more efficient processing can be. However, longer blocks lead to longer reaction times to user input. The size of this block represents a tradeoff between efficiency and latency in response time and is a parameter that is configurable by users of the Subsynth toolkit.

Flyweight Pattern

Subsynth needs to be flexible, but this should not impact its performance. If implemented naively, production and consumption of the `SampleBuffer` blocks could impact the system's performance negatively. There is a design pattern for fast allocation and destruction of large objects called the Flyweight pattern [Flyweight]. In Subsynth, we always allocate and deallocate `SampleBuffer` objects from our own managed memory pool of objects called the `FlyWeightPool` (see Figure 25). This allows the system to reuse objects rather than allocate and deallocate through the system memory manager, a process that can be slow in this case because it is too general.

Subsynth Audio I/O Streams Utility Library

To help facilitate code reuse, we also wrote an audio I/O streams library. These streams are used mainly in the specialized source and sink (output) module types (Figure 26, Figure 27). Our audio streams library was modeled after the standard C++ iostreams interface [IoStream], and it provides a utility library that extensions to the core can use to read or write audio data from a computer's file system or network.

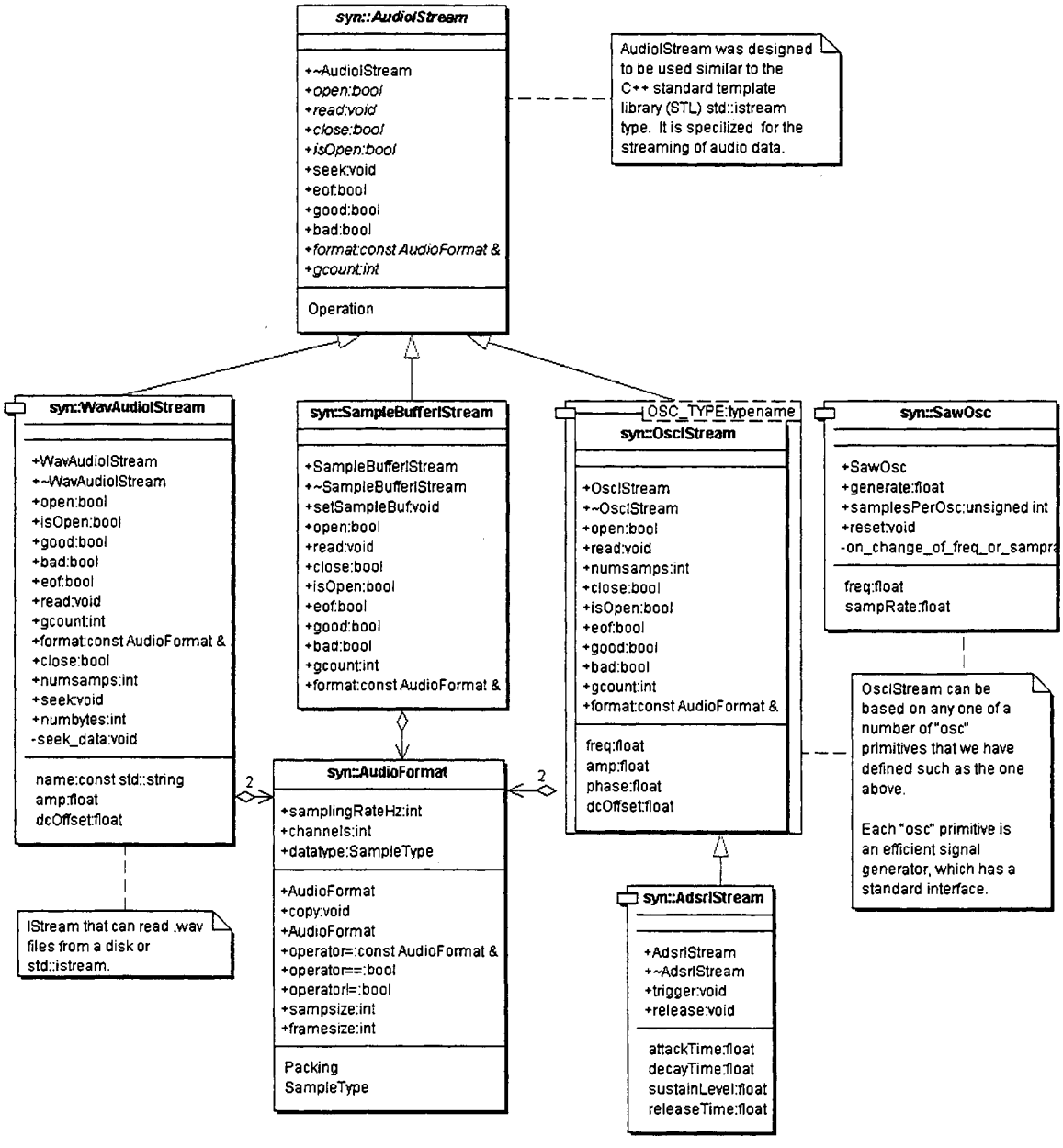

Figure 26. Audio Input Streams [UML].

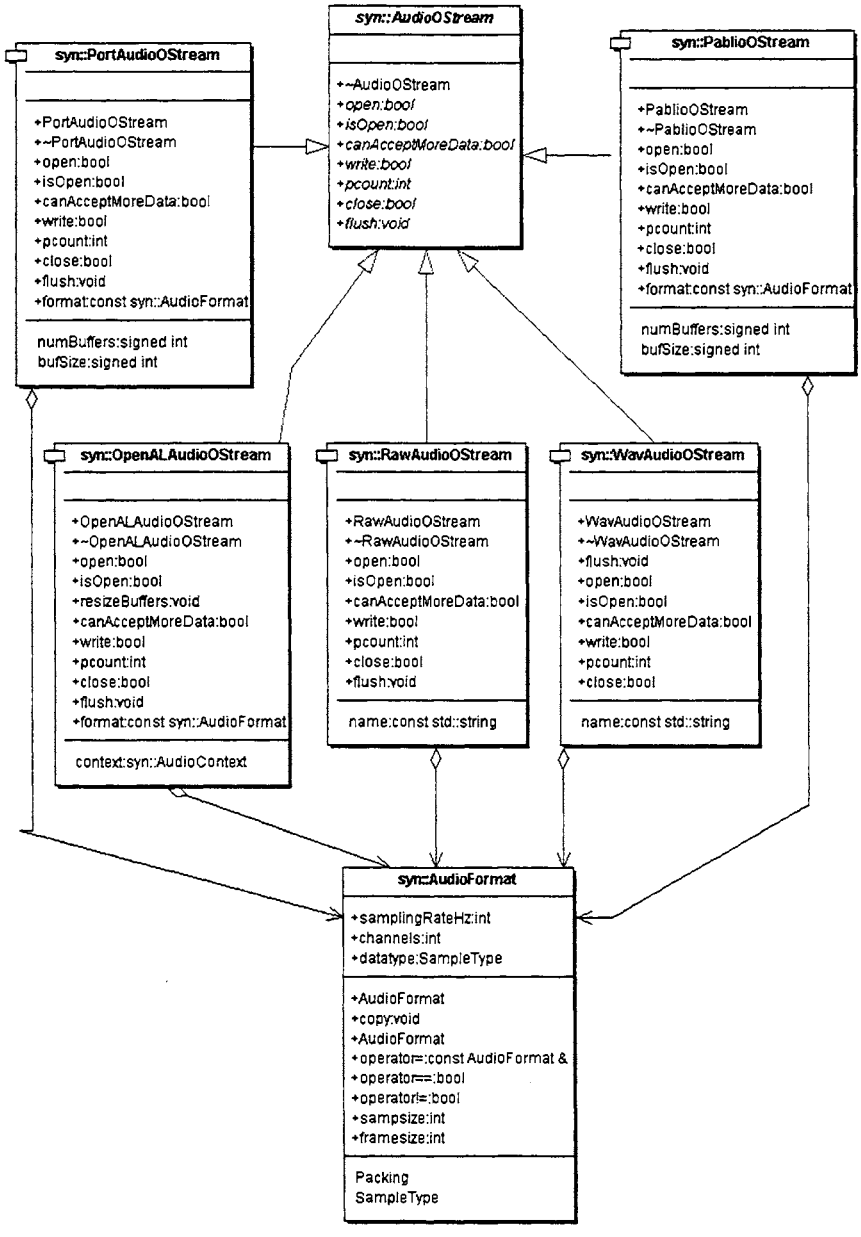

Figure 27. Audio output streams [UML].

Audio Format Utility

We implemented data conversion routines between common audio data formats including any combination of the following five formats: unsigned/signed 8- and 16-bit integer and 32-bit floating point. For speed and generality, we implemented this using template meta-programming [Alexandrescu01], creating generic C++ template code that handles the twenty-five conversion cases

with no loss in runtime performance (Listing 11 Appendix B). With this technique, only the code needed for each conversion case is compiled in. Currently, only the iostreams in Subsynth need conversion since any format may be requested to open each stream, although future additions may take advantage of this utility.

OSC-Concept Library

Some module and stream implementations need access to mathematical functions to generate signals such as noise or sine waves. We took these basic digital oscillator concepts and created a library of them for inclusion by authors of new modules or stream implementations. We call each oscillator (OSC) object a *concept* because it embodies a well-defined independent periodic function. There are seven lightweight OSC-Concept objects currently implemented – triangle, sine, saw, square, white noise, pink noise, and ADSR. Each OSC-Concept is written as a simple standalone object depending only on C++.

Included Modules

We include several unit generators in Subsynth in order to test system performance and baseline functionality. These new specialized types (Figure 28) include:

- **Two digital oscillators.** One for wave-table lookup, and one for procedurally generated functions.
- **Operators for multiplication and summation.** These can be used for modulating a signal's amplitude by another, or for mixing several signals together.
- **High and Low pass Filter.** These remove high or low frequencies from a signal, resulting in thin or muffled sounds.
- **Signal mixer and splitter.** This can add several streams into one, or replicate one stream into many.
- **Audio Stream Adaptor.** This adaptor module can read from and write to any `AudioIOStream`. Currently this means .wav or .raw PCM files are supported. In the future, a network stream could be created and used with this module.
- **OSC-Concept Adaptor.** This adaptor modulè can read from one of the procedural signal generator types in the OSC-Concept library described in the previous section.
- **Envelope generator.** This generates a signal that can be used to provide the effect of attack, decay, sustain, and release when modulated with another audio signal.
- **Sink.** This provides output to audio hardware.

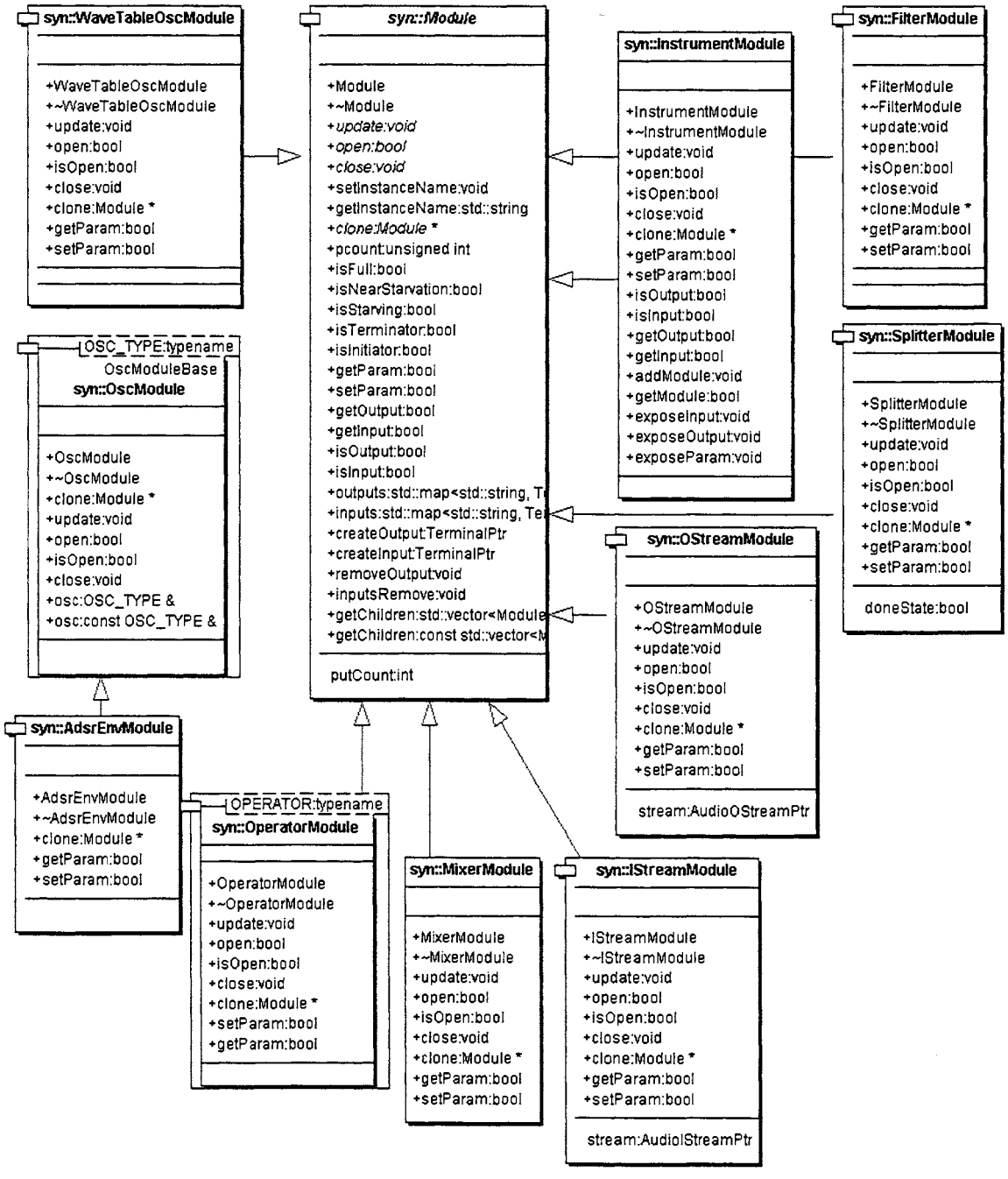

Figure 28. Subsynth Modules [UML].

Using the components included with Subsynth, many audio synthesis methods may be created. For example, the digital oscillators support Wavetable lookup synthesis in which the wavetable can be initialized from either disk, `AudioIStream`, or from one of the `OSC-Concept` generator types

described above. To support frequency modulation synthesis, the digital oscillators support a frequency control signal input. The input works with any `SampleBuffer` audio signal in the range of –1 to 1, and is a linear input (converted to exponential frequency scale 20 – 20000Hz internally). The addition operators facilitate additive synthesis and wavestacking. The multiplication operators can be used to support amplitude modulation synthesis. Subtractive synthesis is possible through any of the filters. These five audio synthesis methods, and combinations of them, offer a large variety of possibilities. New audio synthesis methods are possible by adding new user defined module types to the system. This extensibility is made possible through our generic audio framework design, and is what enables Subsynth to be a general tool to support many audio synthesis methods.

Static control parameters on each module are also available for setting attributes such as amplitude, frequency, triggering, and stopping each sound. Some modules have other static parameters such as envelope or filter cutoff. Depending on the module, static parameters set defaults or interact with the control signal. Parameters are usually how an application will interactively control Subsynth.

Included Builders

Builders are responsible for configuring a network of Subsynth modules. Subsynth defines a one-way dependence of builders to the core audio toolkit (Figure 29). In the figure, the builders (ModuleFactory, and SubsynthInstrumentBuilder) depend on the core audio framework design (Module and InstrumentModule). The benefit to keeping configuration separate from the core is that the user is free to choose existing, or define new, methods of configuration.

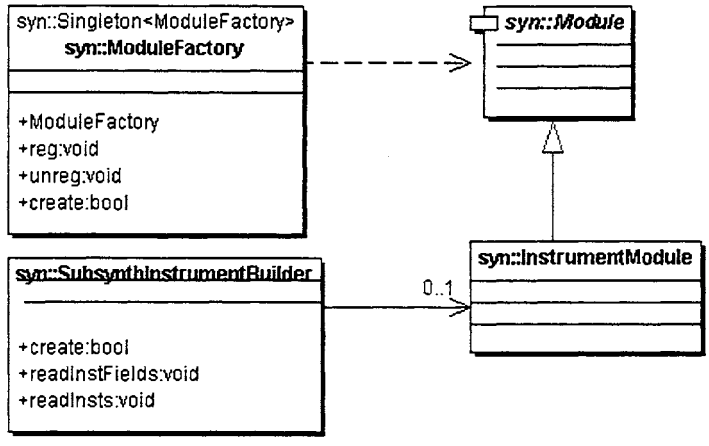

Figure 29. Subsynth Builder UML diagram.

Each builder is responsible for the creation of "instruments". A Subsynth `Instrument` is an aggregate type that provides a façade [Gamma95] to a sub-graph of Subsynth modules. This façade interface is derived from the `Module` interface so that an instrument module will appear the same as any other single module. This way, users can expect to use these complex `Instrument` constructs in their graph designs.

Internally, the instrument contains an arbitrary sub-graph of modules, but the user doesn't need to worry about this, because this is hidden by the façade. The façade (Figure 30) can be configured to selectively expose internal parameters, inputs, and outputs to the module sub-graph contained within. There are two Builder types that Subsynth currently provides to create `Module` objects:

- **SubsynthInstrumentBuilder.** This builder reads an instrument definition data file, which is an XML file format that we have defined that configures an `Instrument`. This format includes ways to add the internal module sub-graph through the addition, configuration, and connection of individual modules. In addition, the format allows configuration of the façade to expose sub-graph parameters, input signals, and output signals. See Listing 10 for an example Subsynth instrument definition.
- **ModuleFactory.** This builder holds a collection of pre-registered modules. To make `ModuleFactory` create a new module, a string is given that matches one of the pre-registered modules. The `SubsynthInstrumentBuilder` makes use of this module so that it can look up known modules from a given string in the XML file.

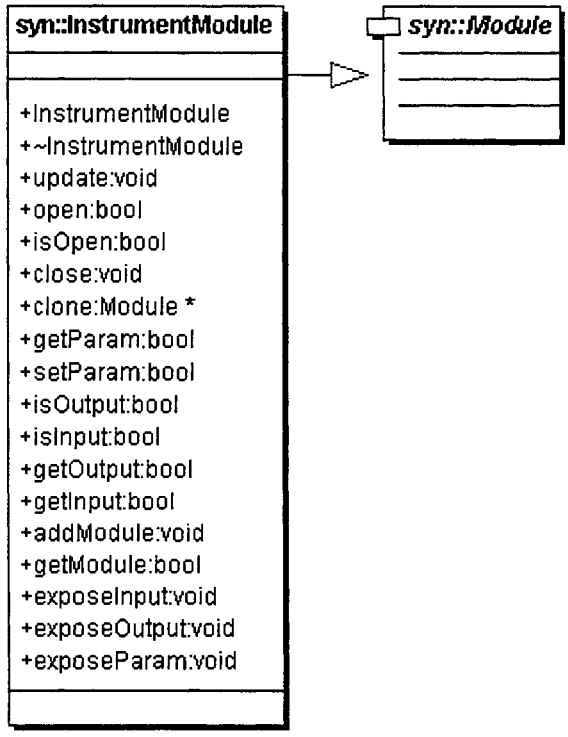

Figure 30. UML diagram for the Instrument type.

Included Runners

Once modules have been created and connected into an audio synthesis graph, they need to be processed regularly. This is done by calling the module's `update()` method (Listing 1). This

method can read data from the module's input terminals, process it in some way, and write it to the output terminals.

Normally, a design of this type is too slow to do signal processing. In Subsynth, audio signals are passed around in blocks of samples (Figure 31), rather than individual samples. This allows us to amortize the cost of function call overhead over many samples. In addition, the `update()` functions have multiple special cases to provide maximum performance for different states that the user can select. For example, some possible cases we optimize for are the usage of audio rate, control rate, or parameter to control the generated audio. Another set of cases we optimize for are in signal generation using no, linear, or cubic interpolation. To illustrate, our `WaveTableOsc` module has four conditional cases, two for control rate, and two for static. Of these, two are for linear interpolation, and two use no interpolation.

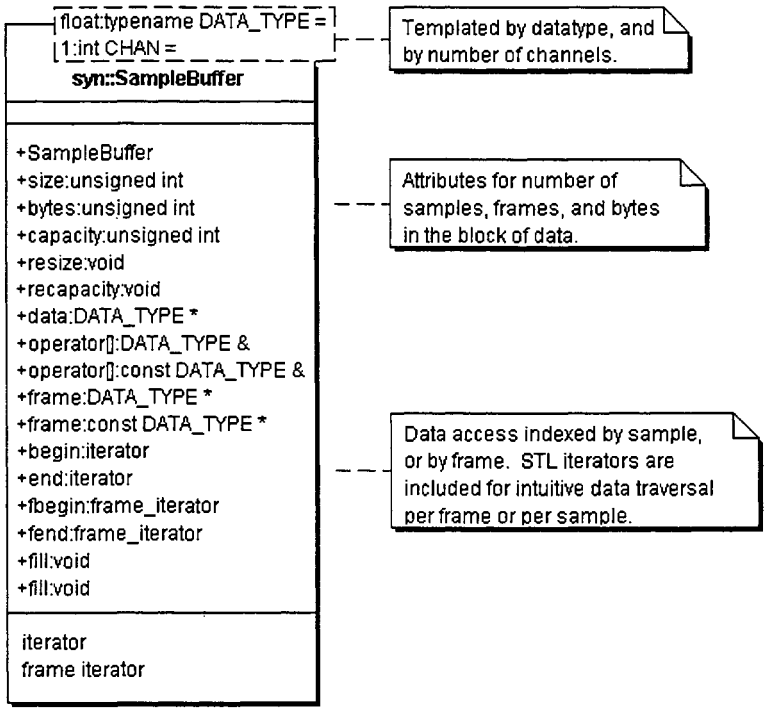

Figure 31. SampleBuffer data type.

If a module is not updated frequently enough, the chain of modules that receive data from it will *starve*. This can cause stuttering or other disruptions to the audio output. This implies the need for a scheduler, something that can produce an optimal execution ordering in which to minimize these starvation cases (given the Subsynth graph topology).

We have provided a container in which to execute the audio synthesis network that we call a "Runner" (Figure 32). A programmer using Subsynth is free to choose between using the provided `Runner` or calling `update()` within their own specialized runner. A `Runner` is responsible for calling `update()` on all the modules in the network in the order that it schedules. To facilitate scalability from low-end to high-end systems, Subsynth's `Runner` uses threads to spread the `update()` execution load across multiple processors. To make the runner flexible, it supports pluggable `Scheduler` objects (Figure 33), which define the scheduling algorithm to use. The scheduling algorithm defines the order of execution, thread, and task dependencies. Subsynth provides a default scheduler that can be replaced with any user-defined implementation. For most customizations, programmers will not need to completely redefine a runner, they can just replace the scheduling algorithm to better fit their needs. Next we discuss a few case studies, which illustrate usage of Subsynth, and some higher-level applications that benefit from Subsynth.

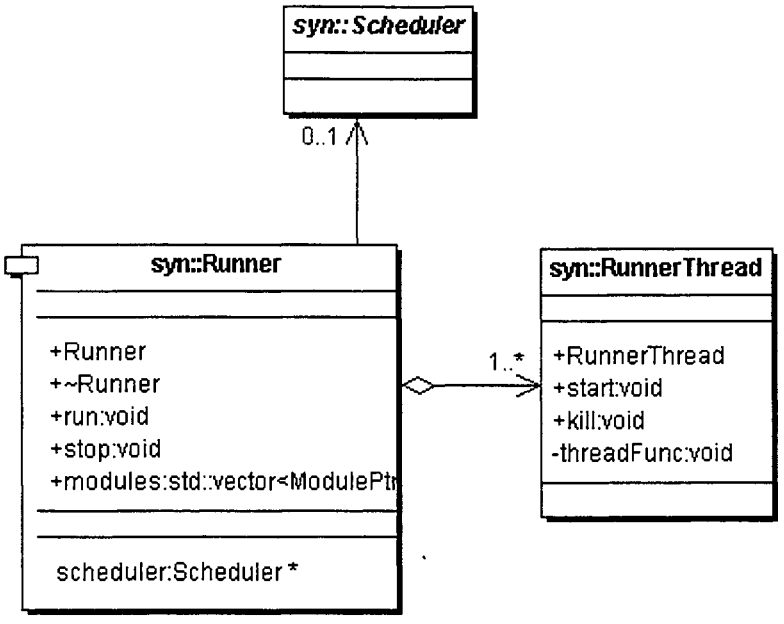

Figure 32. Subsynth task management.

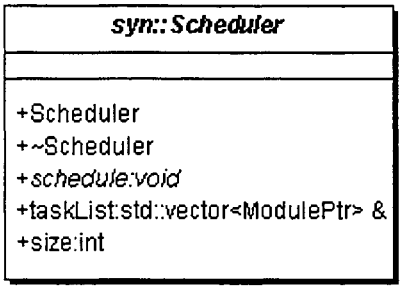

Figure 33. The Scheduler abstract interface.

CHAPTER 6 CASE STUDIES

In this section, we will showcase three tools that have been built upon Subsynth. The purpose of this is to illustrate the effectiveness of Subsynth as a generic audio synthesis subsystem for use by higher-level tools. These higher-level tools we showcase are: SubsynthMIDI, Sonix, and GAME. SubsynthMIDI is a software MIDI interface on top of Subsynth that makes it able to be used by musical applications and also provides a limited sound object representation for sonification. Sonix is a simple sound object library, providing simple control over sampled sound playback. GAME is a music engine employing fractal techniques for generation of music events in response to input data from scientific applications.

SubsynthMIDI

SubsynthMIDI provides a MIDI implementation on top of Subsynth. This interface to Subsynth is intended to enable musical applications. Alternatively, it could be used for sound effects in sonification and virtual environments.

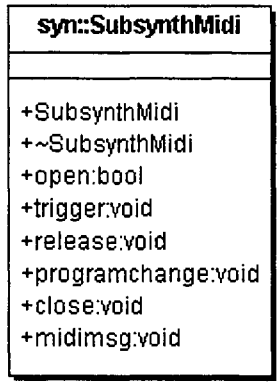

Figure 34. The SubsynthMIDI API.

To provide the ability to switch between the computer's native hardware synthesizer and the Subsynth software synthesizer, a MIDI abstraction was created (Figure 34) that could provide an

interface to either. Several implementations of this interface were then specialized for each operating system, including one for Subsynth.

```
<SubsynthMIDI>
   <instrument channel="1" value="female-voc.inst"/>
   <instrument channel="10" value="timpani.inst"/>
   <instrument channel="3" value="guitar.inst"/>
   <outputport value="pa"/>
   <threads value="1"/>
   <maxpolyphony value="6"/>
   <blocksize value="512"/>
</SubsynthMIDI>
```

Listing 2. Example SubsynthMIDI configuration file.

The `SubsynthMIDI` interface is relatively simple, mirroring the MIDI specification. The internal implementation supports arbitrary polyphony, trigger/release of notes, pitch, channel, and note velocity. MIDI control messages are not implemented yet, but should be trivial to do so. Because Subsynth is flexible, we decided to allow the `SubsynthMIDI` implementation to be configured with an XML file format (Listing 2). This configuration file allows the following options:

- Instrument to use for each of the 16 MIDI channels
- Number of overlapping sounds to play at once (polyphony)
- Number of threads to use for the synthesizer process
- Block size of the sample buffers
- Which audio port on the system to send audio data to.

To test out the MIDI implementation, a music event scheduler called a Sequencer was created. The Sequencer is responsible for dispatching music events to the Subsynth MIDI API. To fill the sequencer with music events, a MIDI file loader was created. This file loader scans a MIDI data file (.mid extension) and sets up the Sequencer appropriately. Using the Sequencer and SubsynthMIDI, we have successfully tested several MIDI files of classical and popular music available on the Internet.

G.A.M.E. Toolkit for Scientific Sonification

G.A.M.E. is a music engine employing fractal techniques for generation of music events in response to input data from scientific applications. This toolkit was needed to allow arbitrary sonification of scientific data, and to enable collaboration with other sonification researchers. The idea was to create a continuous stream of music that reacted with real-time response to the emerging scientific data as the user interacted with it. The goal of this idea was to enable a sort of "wall paper" music that could be pleasant to listen to and would impart changes in mood and hopefully increase perception of the visualized data. This toolkit is called G.A.M.E. to mean "Grammatical Atonal Music Engine". The "grammar" comes from the use of L-System rules, which typically model growth processes [Lsystem]; "atonal" means that we do not make any restrictions on use of pitch. Previous tools mapped data values to specific sets of notes that sound pleasing to most people (in an attempt to create listenable music). The G.A.M.E. system relies on the specification of a grammar to create listenable music. This indirect mapping shows promise where earlier direct mapping efforts produced less desirable results [Bryden02]. This system enables listenable music sonification for many types of scientific data and other applications.

The design has four parts: a generalized L-System framework, an L-System data file loader specialized for XML, a system for parameterization of the real-time application data, and an L-System renderer specialized for SubsynthMIDI. Unlike sonification software that uses MIDI to directly map musical parameters to data, the G.A.M.E. engine creates a music *event stream* via L-System algorithms. Certain events in the event stream can interface with and then respond to the application data. Next, we explain some background on L-Systems, and then present the implementation of G.A.M.E.

L-System Background

An L-System is able to produce fractal patterns modeling growth. Fractal patterns exhibit the growth of increasing detail that is obtained through a procedural definition [Mandelbrot77]. The fractal growth in an L-System is expressed in terms of symbolic elements that are produced by iterating over production rules. The series of elements that are produced are what we call an "L-string". Every L-System is defined by an *axiom* and a set of *production rules*. The axiom is the initial L-string, and the production rules define match and replacement strings that specify how to "grow" that string.

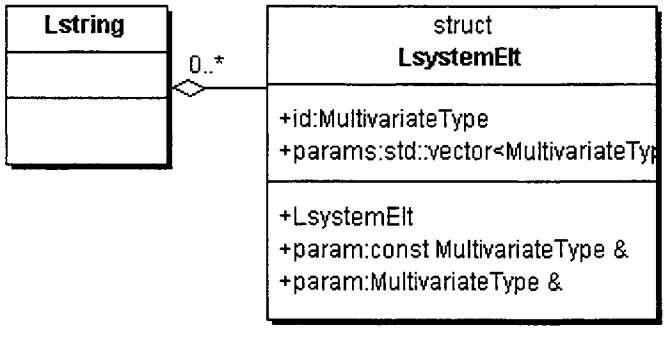

Figure 35. L-System String Element.

In G.A.M.E., each "character" in an L-System string is called an *element*. Each element stores multiple parameters, allowing for grouping of related data (Figure 35). G.A.M.E. uses this parameterized element for use as an *event*. The L-string defines a list of events. Specific to the G.A.M.E system, event types are defined that correspond to music. The resulting string becomes a string of music events, and these music events may be mixed with other procedural events that apply application variables to various music states such as tempo, volume, timbre, and pitch. For example, an event could trigger or release a note while the event's parameters would include pitch, velocity, and the channel to affect.

G.A.M.E. Music Engine Implementation

The `Lsystem` class in G.A.M.E. stores the axiom and production rules (Figure 36). After the class is set up, the user can tell it to apply the rules any number of times to grow the resulting L-string. For music this usually means increasing the complexity or texture of the music.

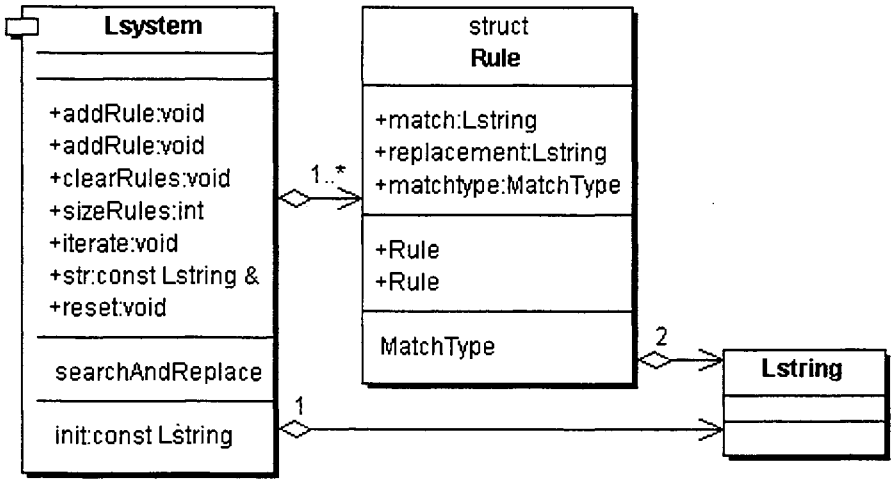

Figure 36. L-System UML Diagram.

The L-system data file format is defined using the XML DTD (XML Schema is also available) format and is constructed with the L-system axiom and a list of production rules (Listing 12). Each production rule has the option of either a regular expression match or an exact match. The "strings" in the format are actually vectors of `<elt>` nodes. Each `elt` node is like a character in a string, except that the `elt` node contains an extra data payload or parameters. This concept is also mirrored in the software. The G.A.M.E. L-System XML format is not tied to music. Because of its general quality, it could also be used for other applications such as graphics. There are predefined element types that we use, however, that match our L-string renderer.

L-System elements are defined as music events. The first ring renderer is an event scheduler that operates on a string of L-System elements (or music events). The renderer turns these events into

MIDI events that are sent to the computer audio device. For the scheduler to work, every element needs to contain at least a command followed by a starting time. The scheduler uses the starting time to determine when to execute the event, and it uses the command tag to determine how to execute it. Once it is executed, the other parameters are read. The renderer can be controlled by the application through a parameter system. These parameters can be referenced in the L-System XML format and then resolved on the fly as each event is executed. This allows application data to influence parameters in the music such as pitch, timbre, volume, and tempo.

Currently, the number of parameters is partly limited by MIDI capabilities. We plan to use a software synthesizer in the future for a greater degree of control. A software synthesizer also will allow this work to run on any computer platform, enabling better collaboration with other researchers who may not have access to similar MIDI hardware.

This technique is useful for selecting production rules based on data defined by the application. This allows a more coarse-grained approach to sonify macro-scale features in the data via the parameter system. This complements the fine-grained control for sonifying micro-scale features with rhythm and motive changes.

Sonix

Sonix provides manipulation of *sound objects* on top of several audio APIs including Subsynth. A sound object provides simple control over playback of sampled sounds. Sound objects are useful to many simple virtual environment applications. The interface to Sonix is kept very simple in order to get users up and running with sound as fast as possible. It allows one to trigger, position, change volume and pitch, or to adjust a filter cutoff on 3D sounds. Systems using this layer expect to be completely portable. Sonix is reconfigurable allowing audio APIs to be safely swapped out at runtime without the dependent systems noticing.

Reconfiguration

Runtime reconfiguration of sound APIs can be useful so that the user can experiment with quality and latency differences of different hardware and sound APIs. If no audio API is available on a given platform, application calls to Sonix are simply ignored. This gives the benefit that no special code application code is needed to enable or disable sound—it is all handled by Sonix.

```
// start sonix using Subsynth
sonix::instance()->changeAPI( "Subsynth" );

// fill out a description for the sound we want to play
snx::SoundInfo sound_info;
sound_info.filename = "808kick.wav";
sound_info.datasource = snx::SoundInfo::FILESYSTEM;

// create the sound object
snx::SoundHandle sound_handle;
sound_handle.init( "my sound for testing" );
sound_handle.configure( sound_info );

// trigger the sound
sound_handle.trigger();
sleep( 1 );

// trigger the sound using a different audio system...
sonix::instance()->changeAPI( "AudioWorks" );
sound_handle.trigger();
sleep( 1 );

// trigger our sound object using different source data
sound_info.filename = "303riff.wav";
sound_handle.configure( sound_info );
sound_handle.trigger();
sleep( 1 );
```

Listing 3. How to reconfigure Sonix at runtime

The benefit of this abstraction is that when something changes, application code does not need to be aware or do any special handling. Everything in Sonix is changeable behind the scenes during application execution. See Listing 3 for an example of how to reconfigure Sonix in C++.

Using Sonix

Sonix only provides wavetable synthesis (see the section on Digital Audio Synthesis above). Even though it is not a complete audio synthesis package, Sonix is still very useful to application

areas such as virtual environments needing access to 3D sound objects. Here, we will provide an example of writing an application using Sonix.

To setup a sound is straightforward as seen in Listing 4. Here, we use a `snx::SoundInfo` object to configure the sound object, which is accessed by a `snx::SoundHandle` object.

```
snx::SoundInfo info;
info.filename = "crack.wav";
info.datasource = snx::SoundInfo::FILESYTEM;

snx::SoundHandle crack_sound( "crack" );
crack_sound.configure( info );
```

Listing 4. Code to setup a Sonix sound

To keep Sonix running, an update function `step( float time_delta )` must be called repeatedly by the application. `time_delta` is the amount of time since `step()` was last called, and `step()` should be called within the application's *frame* function (Listing 4). A frame function is one that is called in an application to update its state, usually many times per second.

```
void frame()
{
   time_delta = getTimeChangeInSeconds(); // use a system call, or
                                          // other API to get your time delta
   sonix::instance()->step( time_delta );
}
```

Listing 5. Call sonix::step() in the application's frame function.

```
#include <iostream>
#include <string>
#include <snx/sonix.h>

int main( int argc, char* argv[] )
{
   std::string filename( "808kick.wav" ), api( "Subsynth" );

   if (!snxFileIO::fileExists( filename.c_str() ))
   {
      std::cout << "File not found: " << filename << "\n" << std::flush;
      return 0;
   }

   // start sonix using Subsynth
   sonix::instance()->changeAPI( api );

   // fill out a description for the sound we want to play
   snx::SoundInfo sound_info;
   sound_info.filename = filename;
   sound_info.datasource = snx::SoundInfo::FILESYSTEM;
```

```
    // create the sound object
    snx::SoundHandle sound_handle;
    sound_handle.init( "my simple sound" );
    sound_handle.configure( sound_info );

    // trigger the sound
    sound_handle.trigger();
    sleep( 1 );

    // trigger the sound from a different position in 3D space...
    sound_handle.setPosition( 10.0f, 0.0f, 0.0f );
    sound_handle.trigger();
    sleep( 1 );

    // this simulates a running application...
    while (1)
    {
       sonix::instance()->step( time_delta );
    }

    return 1;
}
```

Listing 6. Example C++ program that uses Sonix to play a sound using Subsynth.

Sonix Architecture and Design

Sonix was designed to be very simple to use, while offering useful features to general VE applications that need sound. In Figure 37 we see the API for Sonix. The main parts that a programmer will use are shown, namely the `snx::SoundHandle` class, and the `sonix` singleton class. The `sonix` singleton class is used to start, stop and reconfigure the sound system. The `snx::SoundHandle` class is used to manipulate individual sounds. Both classes must be used for any sound to be heard.

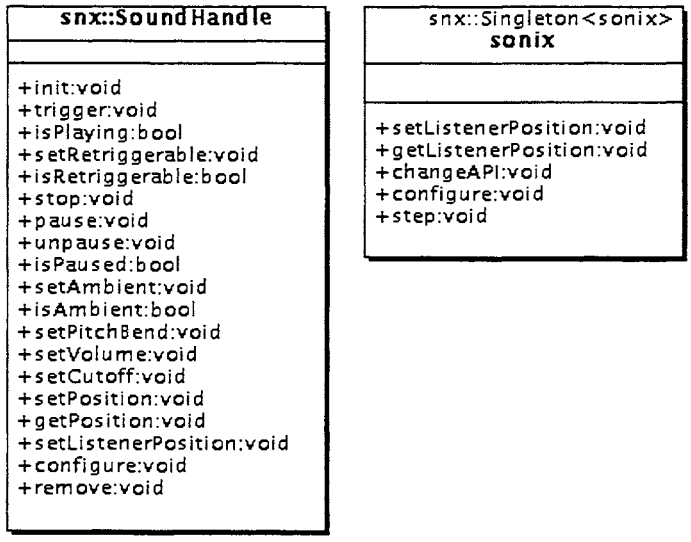

Figure 37. UML diagram of the Sonix application programming interface (API).

Starting the Sonix system is easy (see Listing 7). The call needed to start the system is `changeAPI()`. Currently OpenAL, AudioWorks, and Subsynth toolkits are supported by Sonix. To select one for use, one of the strings "OpenAL", "AudioWorks", or "Subsynth" is passed to `changeAPI()`.

```
sonix::instance()->changeAPI( "Subsynth" );
```

Listing 7. Code to startup and initialize Sonix to use the Subsynth audio subsystem.

Sonix was designed using design patterns and an object-oriented approach (Figure 38). When designing Sonix, we used many design patterns that were appropriate to a simple audio system [GameGems2]. They are as follows:

- **Adapter** (`snx::SoundImplementation`). This adapter provides a common interface to the underlying sound API.
- **Prototype** (`snx::SoundImplementation`). Making `snx::SoundImplementation` a Prototype allows a new cloned object to be created from it that has duplicate state.

- **Plugin Store** (`snx::SoundFactory`). Each sound implementation is registered with a Store called `snx::SoundFactory`. This Store allows users to select items from its inventory. Another name for Store is "Abstract Factory".
- **Abstract Factory** (`snx::SoundFactory`). The Store can create new instances of the requested sound implementation. The Abstract Factory consults its Store of registered objects, and if found, makes a clone of that object (Prototype pattern). The Abstract Factory is used in Sonix to configure the Bridge.
- **Bridge** (`sonix` interface class and `snx::SoundImplementation`). The `sonix` class is the audio system abstraction which is decoupled from its implementation `snx::SoundImplementation`. This way the two can vary independently. Bridge also facilitates run-time configuration of the sound API.
- **Proxy** (`std::string` and `snx::SoundHandle`). `snx::SoundHandle` is how users manipulate their sound object. `snx::SoundHandle` is actually a proxy to a `std::string` proxy. The `std::string` Proxy is what allows Sonix reconfiguration of resources. Rather than using pointers which can easily be left to dangle, the `std::string` serves as a lookup for a protected sound resource located internally to the Sonix tool. The `snx::SoundHandle` wraps this `std::string` to provide a simple and familiar C++ object to use as the sound handle. The `sonix` class acts as Mediator between every Proxy method and the actual audio system Adapter.

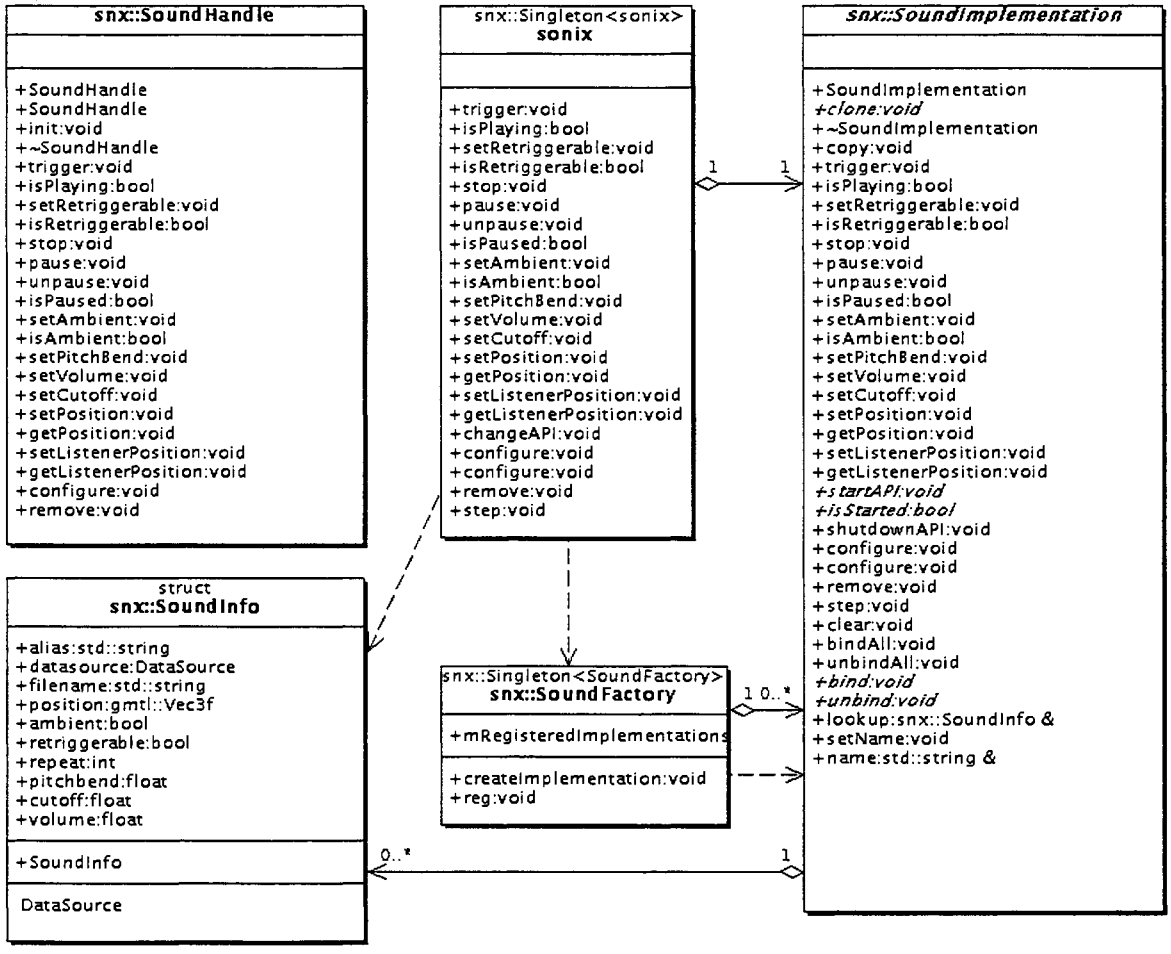

Figure 38. The Sonix sound object system [UML].

Sonix supports the selection of several audio subsystems by the application through implementation plug-ins (Figure 39). Each plug-in implements an adapter to an underlying audio subsystem. The adapter supports a common interface that Sonix knows how talk to. Each adapter is then registered with a factory object, which may ask that adapter to clone itself for use by whoever called the factory.

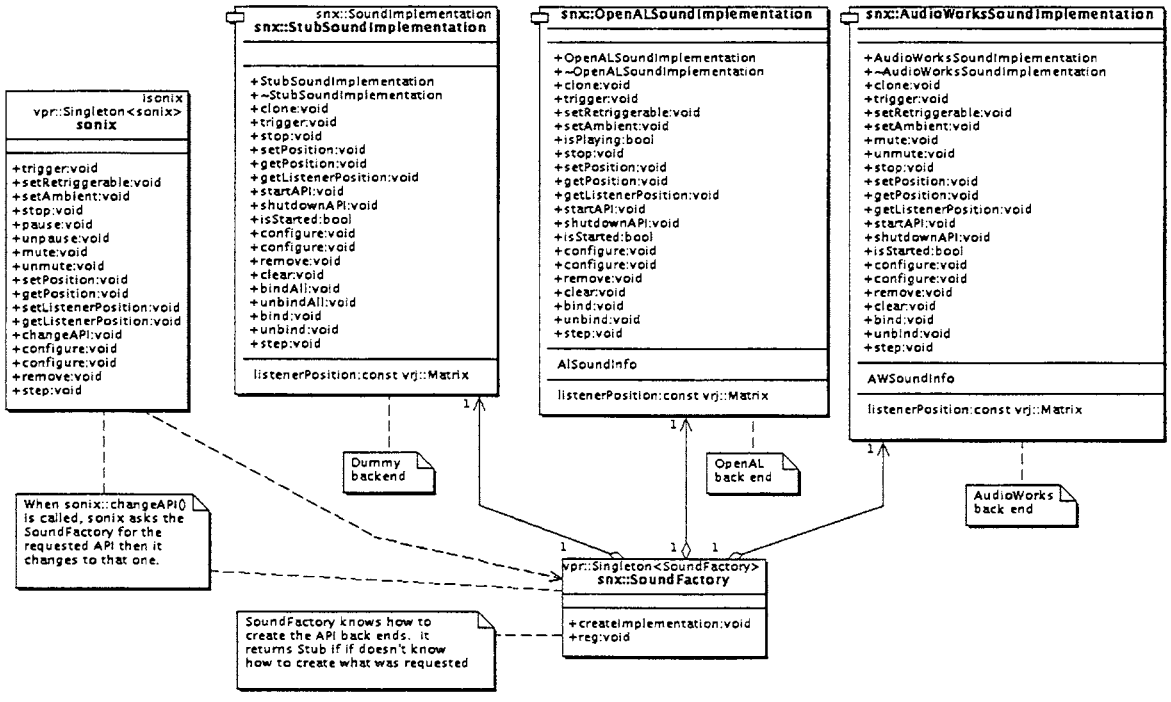

Figure 39. Sonix plugins.

Sonix is an effective tool currently in use by several people at the Virtual Reality Applications Center and by other people external to the center who use the open source VR Juggler toolkit. In the next chapter we evaluate Subsynth's effectiveness as a subsystem for audio synthesis, and how it measures up to our specified requirements.

CHAPTER 7 DISCUSSION OF RESULTS

Benefits

Subsynth has been tested in a small range of applications and it shows good promise as an interactive audio subsystem on which higher-level applications can be based. We have used Subsynth in a virtual environment to sonify events with no perceivable change in graphics frame rate or other critical aspects. We have also used Subsynth as the synthesizer backend for MIDI music, and it performs well. In other words, it meets the following criteria for MIDI:

- Correct pitch for each note is produced.
- Music events occur in a timely and consistent manner, which is very important to accurately reproduce music.
- Voice polyphony of arbitrary number is supported to produce the effect that many instruments are playing together.
- Dynamic range of volume in response to the MIDI velocity parameter.
- Correct timbre that sounds qualitatively "good" to our ear.

Qualitatively, SubsynthMIDI sounds similar to the results produced by a range of other commercial synthesizers.

We have also used Subsynth to build simple sound objects upon in the Sonix toolkit. Sound quality and performance of Subsynth here matches usage of similar tools such as OpenAL. Factors that were noticed were sound quality during pitch bending, frequency filtration, and dynamic range of volume control.

For our last test of Subsynth we have used SubsynthMIDI in the G.A.M.E. toolkit for a new type of scientific sonification based on continuously generating music driven by data. In the future the G.A.M.E. toolkit may benefit from using Subsynth directly for access to more control parameters than offered by MIDI. Currently results in G.A.M.E. with SubsynthMIDI sound similar to using hardware MIDI directly.

Since Subsynth is portable to most computing systems, using MIDI sysex becomes portable. It becomes an option because we can rely on this proprietary feature of MIDI (tied to Subsynth), without worry of it not being available. This is a problem when using proprietary or hardware synthesizers that are not available at every location where the application is run.

Limitations

Proportional to the size of the configured graph, Subsynth requires a percentage of processor power. Previous hardware synthesis solutions operated in external hardware, and did not affect the main CPU. This is a factor to be considered, but we do not see it as a major limitation. Subsynth could be run on a second computer, or on a separate processor in a multi CPU machine. We see the option of running Subsynth in parallel with the main application on the same CPU as an option, not a necessity. This option is valuable because it makes the running of an audio synthesis application potentially easier given that the host CPU can provide enough processing power. For the case studies we presented in this thesis, we found a Pentium III 500Mhz to be too slow to support Subsynth and a graphical OpenGL application at the same time. Meanwhile, we found a Pentium 4 dual 1.5Ghz system to be very adequate to this task. These results were observed with Subsynth compiled with non-optimized code and debugging symbols included. Obviously, the processor requirements have room for decrement once Subsynth is optimized.

While running Subsynth concurrently with the main application may be an immediate shortcoming on some systems, we do not see this as a long-term problem since future machines will be faster. This means that the percentage of the Subsynth processor usage will continuously decrease until it is acceptable by the most demanding applications. In addition, given the possibility to run Subsynth on separate processors, or even separate computers, the application performance hit returns back to the optimal state to be had when hardware synthesizers were used.

A part of Subsynth that needs optimization is the parallel processing runner component in Subsynth. It operates adequately when the number of threads is under the number of processors in the machine, but currently consumes too much CPU due to context switches. Work should be done to minimize thread context switches due to sleeps or yields since they have shown a lot of overhead in our testing. Figure 40 shows our measurements during testing of this phenomenon where, on a Pentium 4 1.5 Gz processor, the time to `yield()` was about 0.02 sec. This time overhead affects time critical code such as streaming of audio to the hardware, or timely scheduling of music events for control of the audio synthesizer.

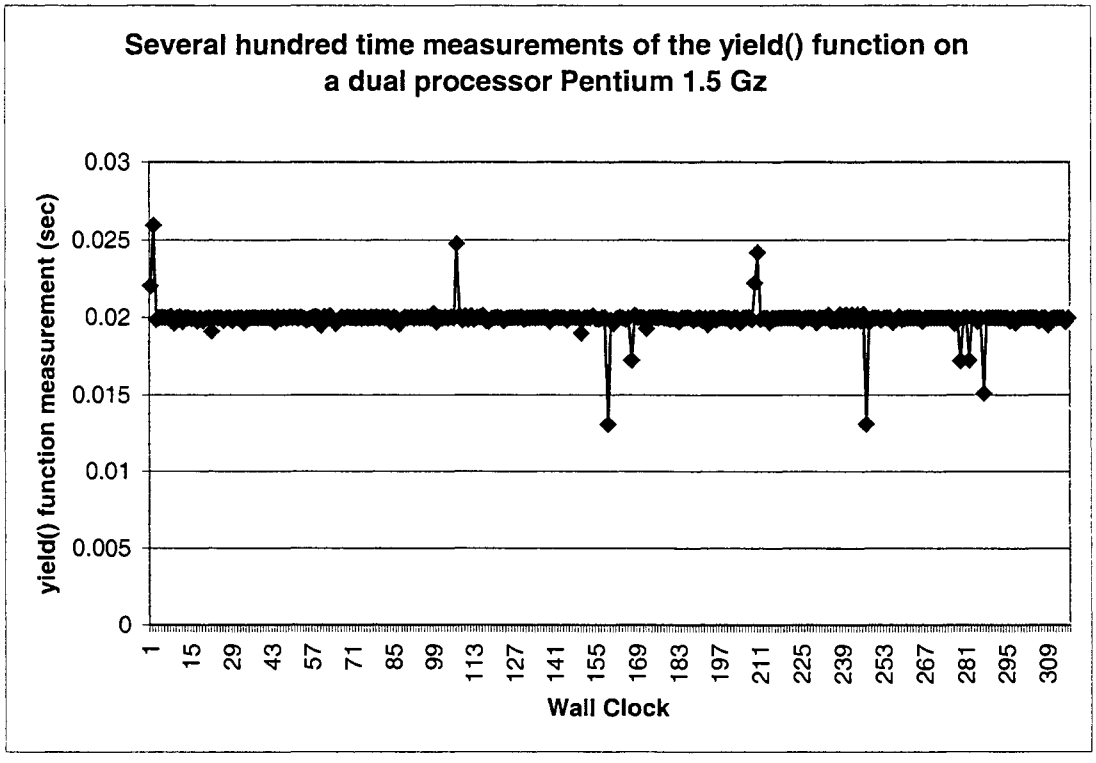

Figure 40. Time measurement of the yield() function.

CHAPTER 8 CONCLUSION

The goal of this work was to address the generality, scalability, and portability problems associated with user-level applications needing real-time audio synthesis. The user-level application

domains we specifically addressed were virtual environments, scientific sonification, and real-time interactive music for composition and performance. We reviewed existing audio synthesis tools, many which of were limited by being too specialized or not portable enough. After this review work, we determined that a new tool was needed. We developed Subsynth, a generic audio synthesis framework for real-time applications. This new tool is meant to enable scalability and portability in higher-level applications and tools that need access to audio synthesis capabilities.

This thesis illustrates the feasibility of developing a generically configurable audio synthesis toolkit that meets the needs of several application domains including virtual environments, interactive composition and performance, and scientific sonification. Subsynth addresses the requirements we have set out for a generic audio synthesis subsystem that would be useful to these real-time interactive applications. Configurability is addressed in the design of the Module, Terminal, Connection framework that allows connection and control of modules in any way. Extendibility is possible in the Subsynth design including the unit generator framework, execution environment, and configuration system. Portability is addressed since Subsynth makes use of cross platform tools and otherwise uses only C++ compliant code. Our case studies showcase that Subsynth has useful synthesis methods, exhibits real-time interactivity, and provides an interface that promotes intuitive control by the application. Because it is implemented in software, Subsynth offers a way to get the same consistent sound across a variety of computing systems. In addition, the tool can be upgraded transparently by simply running it on a computer with more, or more powerful, processors—something not available from most hardware synthesizers. As a proof of concept, Subsynth meets our requirements expectations. As an actual tool, the current status of Subsynth allows it to be usable in a good range of applications, although additional work is required to finalize the implementation of the complete range of features. The next chapter discusses the potential directions, both conceptual and practical, that Subsynth can take.

CHAPTER 9 FUTURE WORK

Subsynth is intended as a long-term research project, which will eventually find its way into a number of virtual reality and multimedia applications. As such, there are a number of directions for future research.

Subsynth needs optimization work done on the multithreading runner. The runner needs a scheduler that can order the task operations more optimally, based on graph dependency and watermarking needs. This sort of work could involve some kind of dynamic scheduler, rather than the included static `Scheduler` design.

Subsynth should provide more unit generators. Particularly lacking are good low-, high-, and band-pass filters. Although Subsynth has acceptable low- and high-pass filters, they are not of good quality and could be improved. For virtual environments, Subsynth especially needs filters for echo and 3D spatialization. For interactive music and sonification, delay and distortion filters could be useful. An authoring tool for instruments would be a useful and short-term project idea to introduce a student to audio synthesis and GUI programming. Additionally, the unit generators need optimizations, such as reducing the number of conditionals that are within inner loops, processing control signals at a reduced rate, and optimizing mathematical functions such as sine and exponential calculation using approximations such lookup tables or processor tricks.

Currently, the Subsynth MIDI implementation is very basic allowing simple MIDI interaction such as pitch, velocity, and trigger. Access to many more parameters is possible through MIDI sysex, and MIDI control messages. Since Subsynth is portable to most computing systems, using sysex becomes an option because we can rely on this proprietary feature of MIDI (tied to Subsynth) without worry of it not being available.

Finally, more applications need to be written on top of Subsynth to promote development and to work out functionality and usability flaws. Obvious application types include those from the domains

laid out in the requirements section. Additionally, we have some specific ideas of research tools that we particularly feel would be interesting to develop with Subsynth:

- **Modulation Synthesis Parameter Selection Using Brute Force Techniques**: Earlier, we explained how modulation synthesis could be used to create very simple instruments that yield very complex sound. The drawback with this technique, as we explained, was that that the parameters to specify the modulation synthesis instrument are unintuitive and hard to get right. Therefore, we feel that it would be interesting to use the brute-force techniques described by the field of artificial-life [Ashlock02]. These techniques have the ability to evolve the parameters of modulation synthesis instruments in a controlled way to converge upon an optimal solution. This best solution would allow us to achieve a closer match to real world instruments using these computationally inexpensive unit generators. We hypothesize that the fitness function could be based on the spectra (frequencies) found in the source and destination waveforms.

- **Visual Authoring Tools for Synthesis Networks**: With the interactive realities becoming more available to average consumers through console gaming systems and more advanced VR systems, authoring tools will become necessary for people to customize "their piece of the Internet." This is similar to how HTML and web browsers have enabled mass participation with the current Internet. With access to more advanced equipment for less money, there will be a growing need to improve interaction techniques with virtual worlds. One especially useful application will be simple intuitive authoring tools that offer these powerful manipulation techniques [RealtimeVR][Otherland]. Sound will be a part of these authoring tools. Users will need to set up a range of sonification methods from this tool. Primarily, it should support creation of sounds and the ability to link them to triggers placed in the virtual environment. Sound attributes such as

frequency, volume, and timbre, should be available to modify and link to other triggers. Specific sound and music research tools could also be made. Subsynth provides a very flexible audio backend to support these needs. It would be interesting to use Subsynth to enable some of these virtual environment authoring tools.

In summary, there is much potential to build upon a generic audio synthesis tool such as Subsynth. A tool such as this could open the door to many research topics while remaining a very generic audio toolkit that creative applications could exploit.

Appendix A

```
;*****************************************************************
instr 1   ; Table Based Rezzy Synth
          ; [from http://www.csounds.com/mikelson/ (6.6.2002)]

idur   = p3
iamp   = p4
ifqc   = cpspch(p5)
irez   = p7
itabl1 = p8

; Amplitude envelope
kaenv  linseg 0, .01, 1, p3-.02, 1, .01, 0

; Frequency Sweep
kfco linseg .1*p6, .5*p3, p6, .5*p3, .1*p6

; This relationship attempts to separate Freq from Res.
ka1 = 100/irez/sqrt(kfco)-1
ka2 = 1000/kfco

; Initialize Yn-1 & Yn-2 to zero
aynm1 init 0
aynm2 init 0

; Oscillator
  axn oscil iamp, ifqc, itabl1

; Replace the differential eq. with a difference eq.
  ayn = ((ka1+2*ka2)*aynm1-ka2*aynm2+axn)/(1+ka1+ka2)
  aynm2 = aynm1
  aynm1 = ayn

; Amp envelope and output
  aout = ayn * kaenv
  out aout

endin
```

Listing 8. Example of an instrument defined in CSound's instrument definition scripting language.

```
(
// synthetic piano 2: from www.audiosynth.com (6.6.2002)
var n, keynote;
n = 6; // number of keys playing
keynote = 12.rand;
Synth.play({
 var z;
 z = Mix.arFill(n, { // mix an array of notes
  var delayTime, pitch, detune, strike, hammerEnv, hammer;

  // calculate delay based on a random note
  pitch = [0,2,4,5,7,9,11].choose + [0,12,24,36].choose + 36 + keynote;
  strike = Dust.ar(0.2+0.4.rand, 0.1); // random period for each key
  hammerEnv = Decay2.ar(strike, 0.008, 0.04); // excitation envelope
  Pan2.ar(
   // array of 3 strings per note
   Mix.ar(Array.fill(3, { arg i;
    // detune strings, calculate delay time :
    detune = #[-0.05, 0, 0.04].at(i);
    delayTime = 1 / (pitch + detune).midicps;
    // each string gets own exciter :
```

```
    hammer = LFNoise2.ar(3000, hammerEnv); // 3000 Hz was chosen by ear..
    CombL.ar( hammer,      // used as a string resonator
              delayTime,   // max delay time
              delayTime,   // actual delay time
              6 )          // decay time of string
   })),
   (pitch - 36)/27 - 1 // pan position: lo notes left, hi notes right
  )
 });
 4.do({
  z = AllpassN.ar(z, 0.040, [0.040.rand,0.040.rand], 8)
 });
 z
})
)
(
// synthetic piano 3
var n, keynote;
n = 10; // number of keys playing
keynote = 12.rand;
Synth.play({
 var z;
 z = Mix.arFill(n, { // mix an array of notes
  var delayTime, pitch, detune, strike, hammerEnv, hammer;

  // calculate delay based on a random note
  pitch = [0,2,4,5,7,9].choose + [0,12,24,36].choose + 36 + keynote;
  strike = Dust.ar(0.1+0.2.rand, 0.1); // random period for each key
  hammerEnv = Decay2.ar(strike, 0.015, 0.04); // excitation envelope
  Pan2.ar(
   // array of 3 strings per note
   Mix.ar(Array.fill(3, { arg i;
    // detune strings, calculate delay time :
    detune = #[-0.05, 0, 0.04].at( i );
    delayTime = 1 / (pitch + detune).midicps;
    // each string gets own exciter :
    hammer = LFNoise2.ar( 3000, hammerEnv ); // 3000 Hz was chosen by ear..
    CombL.ar( hammer,      // used as a string resonator
              delayTime,   // max delay time
              delayTime,   // actual delay time
              6 )          // decay time of string
   })),
   (pitch - 36)/27 - 1 // pan position: lo notes left, hi notes right
  )
 });
 6.do({
  z = AllpassN.ar(z, 0.040, [0.040.rand,0.040.rand], 8)
 });
 z
})
)
```

Listing 9. Example of a SuperCollider script.

```
<?xml version="1.0"?>
<instruments>
   <instrument name="wave sample player">
      <module type="WaveTableOsc" name="Tone">
         <setparam name="filename" value="snare-singleshot.wav"/>
         <setparam name="samplebased" value="1"/>
         <setparam name="basefreq" value="440"/>
         <setparam name="loop" value="0"/>
         <setparam name="interp" value="1"/>
         <setparam name="retrig" value="1"/>
         <setparam name="freq" value="440"/>
         <setparam name="freqcontrol" value="1"/>
         <setparam name="freqcontrolsens" value="1"/>
         <setparam name="trigger" value="1"/>
      </module>
      <module type="AdsrEnv" name="ADSR1"/>
      <module type="Mult" name="Mult1"/>
      <module type="Mult" name="Atten"/>

      <connection>
         <output name="mono audio" module="Tone"/>
         <input name="mono audio0" module="Mult1"/>
      </connection>
      <connection>
         <output name="mono audio" module="ADSR1"/>
         <input name="mono audio1" module="Mult1"/>
      </connection>
      <connection>
         <output name="mono audio" module="Mult1"/>
         <input name="mono audio0" module="Atten"/>
      </connection>

      <exposeoutput name="mono audio">
         <output name="mono audio" module="Atten"/>
      </exposeoutput>

      <exposeparam name="velocity">
         <param name="constant" module="Atten"/>
      </exposeparam>
      <exposeparam name="freq">
         <param name="freq" module="Tone"/>
      </exposeparam>
      <exposeparam name="freqcontrol">
         <param name="freqcontrol" module="Tone"/>
      </exposeparam>
      <exposeparam name="freqcontrolsens">
         <param name="freqcontrolsens" module="Tone"/>
      </exposeparam>
      <exposeparam name="trigger">
         <param name="trigger" module="ADSR1"/>
         <param name="trigger" module="Tone"/>
      </exposeparam>
   </instrument>
</instruments>
```

Listing 10. An example of a Subsynth instrument definition file.

Appendix B

```
namespace ac
{
   struct InIsLarger
   {
      template <typename _in, typename _out>
      static void execute( const _in& i, _out& o )
      {
         const _in in_range = (_in)(((float)audio_data_traits<_in>::range) /
                                                                  (float)4);
         const _in in_min = (_in)audio_data_traits<_in>::min;
         const _out out_range =
             (_out)(((float)audio_data_traits<_out>::range) / (float)4);
         const _out out_min = (_out)audio_data_traits<_out>::min;

         // offset by <in>::min, then scale, then offset back by <out>::min
         const _in scale = ((_in)out_range) / in_range;
         const _in inv_scale = in_range / ((_in)out_range);
         if (scale != 0.0) // float to int (float mult, 2 adds)
            o = (_out) (((i - in_min) * scale) + out_min);
         else              // 16 to 8 bit (shift right, 2 adds)
            o = (_out) (((i - in_min) / inv_scale) + out_min);
      }
   };

   struct OutIsLarger
   {
      template <typename _in, typename _out>
      static void execute( const _in& i, _out& o )
      {
         const _in in_range = (_in)(((float)audio_data_traits<_in>::range) /
                                                  (float)4);
         const _in in_min = (_in)audio_data_traits<_in>::min;
         const _out out_range =
                (_out)(((float)audio_data_traits<_out>::range) / (float)4);
         const _out out_min = (_out)audio_data_traits<_out>::min;

         const _out scale = out_range / ((_out)in_range);
         SYN_STATIC_ASSERT( scale != (_out)0.0 );

         // 8 to 16 bit conversion (shift left, 2 adds)
         // integer to float (float mult, 2 float adds)
         o = (_out) (((i - in_min) * scale) + out_min);
      }
   };

   struct Adjust
   {
      template <typename _in, typename _out>
      static void execute( const _in& i, _out& o )
      {
         if (i >= 0.9999f)
            o -= (_out)1;
      }
   };

   struct DoNothing
   {
      template <typename a, typename b>
      static void execute( a aa, b bb ) {}
   };

   struct Copy
   {
      template <typename _in, typename _out>
```

```
      static void execute( const _in& i, _out& o ) { o = (_out)i; }
   };
   // some flags needed by audio_convert
   template <typename in, typename out>
   struct Flags
   {
      // are the two datatypes the same?
      static const bool types_are_the_same;

      // is the 1st one float and the 2nd one int?
      static const bool float_to_int;
   };

   // until type_info::operator==() works statically,
   // this will have to do...
   template <typename in, typename out>
   const bool Flags<in,out>::types_are_the_same =
      audio_data_traits<in>::max == audio_data_traits<out>::max;

   template <typename in, typename out>
   const bool Flags<in,out>::float_to_int =
      audio_data_traits<in>::range == audio_data_traits<float>::range &&
      audio_data_traits<out>::range != audio_data_traits<float>::range;
}

/** convert one audio sample to a sample of a different format
 *  currently works for any type of data supported by audio_data_traits.
 */
template <typename in, typename out>
inline void audio_convert( const in& i, out& o )
{
   // in == out: Copy,  in > out: InIsLarger, out > in: OutIsLarger
   meta::IF<ac::Flags<in,out>::types_are_the_same, ac::Copy,
      /*else*/ meta::IF<sizeof( in ) >= sizeof( out ), ac::InIsLarger,
      /*else*/ ac::OutIsLarger>::RET >
      ::RET::execute( i, o );

   // degenerate case: converting float(1.0) to int yields 0 (incorrect), fix
   meta::IF<ac::Flags<in,out>::float_to_int, ac::Adjust,
      /*else*/ ac::DoNothing>
      ::RET::execute( i, o );
}
```

Listing 11. Template metaprogramming audio format conversion routine.

Appendix C

```
<?xml version="1.0" ?>
<!DOCTYPE article PUBLIC "-//vrac//lsystem V0.1.2//EN"
  "http://www.vrac.iastate.edu/~kevn/lsystem.dtd">
<root>
   <lsystem>
      <axiom>
         <lstring>
            <!-- strings are composed of elts, not characters
                 an "elt" is a sort of "super character".
              -->
            <elt name="TEMPO">
               <param id="0">0</param>
               <param id="1">${cg}</param>
            </elt>
            <elt name="DUR">
               <param id="0">0</param>
               <param id="1">${cg}</param>
            </elt>
            <elt name="PROGCHANGE">
               <param id="0">0</param>
               <param id="1">0</param>
               <param id="2">25</param>
            </elt>
            <elt name="NOTE">
               <param id="0">0</param>
               <param id="1">60</param>
               <param id="2">0</param>
               <param id="3">0.7</param>
            </elt>
            <elt name="NOTE">
               <param id="0">${DUR}</param>
               <param id="1">60</param>
               <param id="2">0</param>
               <param id="3">0</param>
            </elt>
            </lstring>
      </axiom>
      <!-- define replacement rules -->
      <rules>
         <rule type="exact"> <!-- can specify regex or exact matching -->
            <!-- for every place in the base string that matches this -->
            <match>
               <lstring>
                  <elt name="NOTE">
                     <param id="0">0</param>
                     <param id="1">62</param>
                     <param id="2">0</param>
                     <param id="3">${t}</param>
                  </elt>
                  <elt name="NOTE">
                     <param id="0">${DUR}</param>
                     <param id="1">62</param>
                     <param id="2">0</param>
                     <param id="3">0</param>
                  </elt>
               </lstring>
            </match>
<!-- ...replace it with this -->
            <replace>
               <lstring>
                 <elt name="DUR*">
                     <param id="0">0</param>
                     <param id="1">0.5</param>
                  </elt>
```

```
                <elt name="NOTE">
                   <param id="0">0</param>
                   <param id="1">62</param>
                   <param id="2">0</param>
                   <param id="3">${t}</param>
                </elt>
                <elt name="NOTE">
                   <param id="0">${DUR}</param>
                   <param id="1">62</param>
                   <param id="2">0</param>
                   <param id="3">0</param>
                </elt>
                <elt name="NOTE">
                   <param id="0">0</param>
                   <param id="1">58</param>
                   <param id="2">0</param>
                   <param id="3">${a}</param>
                </elt>
                <elt name="NOTE">
                   <param id="0">${DUR}</param>
                   <param id="1">58</param>
                   <param id="2">0</param>
                   <param id="3">0</param>
                </elt>
                <elt name="DUR*">
                   <param id="0">${a}</param>
                   <param id="1">2</param>
                </elt>
             </lstring>
          </replace>
       </rule>
    </rules>
 </lsystem>
</root>
```

Listing 12. Example music L-system, using the L-system XML schema specified by G.A.M.E.

BIBLIOGRAPHY

[AiffSpec] Audio Interchange File Format, http://www.borg.com/~jglatt/tech/aiff.htm, verified June 06 2002

[Alexandrescu01] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley Professional, 2001

[Apache] Apache Software Foundation, http://www.apache.org, verified August 5 2002

[Ashlock02] Evolutionary Computation for Modeling and Optimization Online Lecture Notes http://www.math.iastate.edu/danwell/ma378/ln378.html, verified June 06 2002

[ASIO] ASIO Developer Website, http://www.steinberg.net/developers/ASIO2SDKAbout.phtml, verified June 06 2002

[AuditoryDisplay] G. Kramer, *Auditory Display*, "Auditory display of computational fluid dynamics data", p327

[AuditoryIcons] G. Kramer, *Auditory Display*, W. Gaver, "Using and creating auditory icons", p417

[Balking] M. Grand, "Balking Pattern", http://www.clickblocks.org/patterns1/pattern_synopses.htm#Balking, verified June 06 2002

[Bryden02] K. Bryden, K. Meinert, D. Ashlock, and K. Bryden. "Transforming Data into Music Using Fractal Algorithms", paper to be presented at ANNIE 2002, St. Louis, Missouri, November 10-13.

[Cake] Twelvetone Systems, "Cakewalk Music Software", http://www.cakewalk.com, verified August 5 2002

[Chowning73] J. Chowning. C. Roads. J. Strawn. "The synthesis of complex audio spectra by means of frequency modulation.". "Journal of the Audio Engineering Society". Foundations of computer music (reprinted in C. by Roads, Strawn). CambridgeMA. 1973. MIT Press.

[CppDOM] CppDOM Toolkit Homepage, http://xml-cppdom.sf.net, verified June 06 2002

[Creative] Creative Labs Open Source Website, http://opensource.creative.com/, verified June 06 2002

[CreativeDev] Creative Labs Developer Website, http://developer.creative.com/, verified June 06 2002

[CSound] Massachusetts Institute of Technology, "CSound audio synthesis program". CSound Website, http://www.csounds.com/, verified June 06 2002.

[CsoundRef] The Public CSound Manual: Version 4.16, http://www.lakewoodsound.com/csound/hypertext/manual.htm, verified June 06 2002.

[Deatherage72] B. Deatherage, "Auditory and Other Sensory Forms of Information Presentation." In Van Cott and Kinkade (Eds.), Human Engineering Guide to Equipment Design, Revised Ed., Washington: U.S. Gov't Printing Office, 1972, p. 124.

[Dodge97] C. Dodge, T. Jerse, *Computer Music*, Second Edition, Schirmer Books, 1997

[Earcons] M. Blattner, A. Papp, E. Glinert, "Sonic Enhancement of two dimensional graphics displays", G. Kramer, *Auditory Display*, p447

[Flyweight] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, pp 195, Addison-Wesley, 1995

[Fouad00] H. Fouad, J. A. Ballas, D. Brock, "An extensible toolkit for creating virtual sonic environments", in `Proceedings of International conference on auditory displays 2000', ICAD, Atlanta, Georgia, USA.

[Fruityloops] Image-Line Software, "Fruityloops music composition tool", http://www.fruityloops.org, verified June 06 2002.

[GameGems1] M. DeLoura, *Game Programming Gems*, Charles River Media, 2000

[GameGems2] K. Weiner, "Interactive Processing Pipeline for Digital Audio", *Game Programming Gems 2*, pp 529-538. Charles River Media, Inc.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, 1995

[Genome] The Human Genome Project - http://www.ornl.gov/hgmis/, verified June 06 2002.

[GuardedSuspension] M. Grand, "Guarded Suspension Pattern", http://www.clickblocks.org/patterns1/pattern_synopses.htm#Guarded%20Suspension, verified June 06 2002.

[HRTF] K. Pimentel, K. Teixeira, *Virtual Reality Through the New Looking Glass*, Second Edition, pp 168, Windcrest McGraw-Hill, 1995

[Id] Id Software, http://www.idsoftware.com, verified August 10, 2002

[Interp] C. Roads. *The Computer Music Tutorial.* "Waveform Segment Techniques", p. 320, MIT Press. 1996. 81. Copyright © 1996 Massachusetts Institute of Technology

[IoStream] C++ iostream library reference, http://www.cplusplus.com/ref/iostream/

[Kaper98] H. Kaper, S. Tipei, "Manifold Compositions, Music Visualization, and Scientific Sonification in an Immersive Virtual-Reality Environment." In the proceedings of the Int'l Computer Music Conference '98, Ann Arbor, Michigan (October 1998), pp. 399-405

[Kernighan88] B. Kernighan, D. Ritchie, *C Programming Language (2nd Edition)*, Prentice Hall, 1988

[Kramer94] G. Kramer, *Auditory Display: Sonification, Audification, and Auditory Interfaces*, Perseus Publishing, 1994

[Kramer97] G. Kramer. et al. "Sonification Report: Status of the Field and Research Agenda". 1997. http://www.icad.org/websiteV2.0/References/nsf.html, verified June 06 2002.

[Lsystem] P. Prusinkiewicz, A. Lindenmayer, P. Prusinkiewicz, M. Cutter, *The Algorithmic Beauty of Plants*, Springer Verlag, 1990

[Mandelbrot77] B. Mandelbrot, *Fractals: Form, Chance and Dimension*, 1977

[Manimaran01] G. Manimaran, C. Siva Ram Murthy, *Resource Management in Real-Time Systems and Networks*, p 4, MIT Press, 2001

[Mathews69] M. Mathews. "The technology of computer music." Proceedings of ACM Symposium on User Interface Software and Technology. Cambridge MA. 1969. MIT Press.

[MIDISpec] MIDI Specification, http://www.borg.com/~jglatt/tech/midispec.htm, verified June 06 2002.

[Moore98] F. Moore, *Elements of Computer Music*, Prentice Hall, 1998

[MP3Fraunhofer] Fraunhofer MPEG Layer 3 Info, http://www.iis.fhg.de/amm/techinf/layer3/, verified June 06 2002.

[MP3Spec] Inside the MP3 Codec, http://www.mp3-converter.com/mp3codec/, verified June 06 2002.

[MPI] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1999

[Murray96] J. Murray, W. VanRyper, *Graphics File Formats*, p. 179, O'Reilly & Associates, 1996

[NSPR] The Netscape Portable Runtime, http://www.mozilla.org, verified June 06 2002.

[OpenAL] Creative Labs, Loki Entertainment, "The OpenAL audio API", http://www.openal.org, verified June 06 2002.

[Ogg] Ogg Website, http://www.vorbis.com/, verified June 06 2002.

[OpenSG] OpenSG Website, http://www.opensg.org, verified June 06 2002.

[OSI] Open Source Initiative, http://www.opensource.org, verified June 06 2002.

[Otherland] T. Williams, *Otherland: Volume One, City Of Golden Shadow*, Daw Books, Inc, 1997

[Pausch92] R. Pausch, T. Crea, M. Conway, "A Literature Survey for Virtual Environments: Military Flight Simulator Visual Systems and Simulator Sickness." Presence 1(3), pp 344-363, MIT Press, 1992

[Pierce92] J. Pierce. *The science of musical sound* (2nd ed.), New York. 1992. W. H. Freeman and Company.

[PortAudio] PortAudio Website, http://www.portaudio.com/, verified June 06 2002.

[ProducerConsumer] M. Grand, "Producer/Consumer Pattern", http://www.clickblocks.org/patterns1/pattern_synopses.htm#Producer-Consumer, verified June 06 2002.

[Propellerheads] Propellerheads, "Reason and Rebirth music composition tools", http://www.propellerheads.se mozilla NSPR website

[RealtimeVR] K. Pimentel, K. Teixeira, *Virtual Reality Through the New Looking Glass*, Second Edition, pp 108-109, Windcrest McGraw-Hill, 1995

[Roads96] C. Roads. *The Computer Music Tutorial.* MIT Press. 1996. 81. Copyright © 1996 Massachusetts Institute of Technology.

[Rockstar] Rockstar Games, http://www.rockstargames.com/, verified August 10, 2002

[Sonification] K. Pimentel, K. Teixeira, *Virtual Reality Through the New Looking Glass*, Second Edition, pp 98-100, Windcrest McGraw-Hill, 1995

[SCEI] Sony Computer Entertainment Inc., *Ico*, http://www.icothegame.com/en_GB/index.htm, verified August 10, 2002

[Steiglitz96] K. Steiglitz, *A DSP Primer : With Applications to Digital Audio and Computer Music*, Addison-Wesley Pub Co, 1996

[Stroustrup00] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Pub Co, 2000

[SuperCollider] SuperCollider Website, http://www.audiosynth.com/, verified June 06 2002.

[UML] Unified Modeling Language, http://www.omg.org/uml/, verified August 5 2002

[VBAP] V. Pulkki, "Virtual Source Positioning Using Vector Base Amplitude Panning", Journal of the Audio Engineering Society 45, 6 (1997), 456-466.

[Visualization] A. Watt, M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, p. 297, Addison Wesley, 1995

[VRJuggler] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A Virtual Platform for Virtual Reality Application Development". Published at IEEE VR 2001, Yokohama, March 2001.

[VRAC] Virtual Reality Applications Center, http://www.vrac.iastate.edu, verified August 10, 2002

[VSS] NCSA Sound Server Reference Manual, http://www.isl.uiuc.edu/software/vss_reference/vss3.0ref.html, verified June 06 2002.

[WavSpec] WAVE File Format, http://www.borg.com/~jglatt/tech/wave.htm, verified June 06 2002.

[XML] Extensible Markup Language (XML), http://www.w3.org/XML/, verified August 5, 2002